UNISYS

B 2000/B 3000/ B 4000/V Series BPL Compiler

Programming Reference Manual

Relative to Release Level 7.2

Priced Item

August 1987
Distribution Code SD
Printed in U S America
5024789

UNISYS

B 2000/B 3000/ B 4000/V Series BPL Compiler

Programming Reference Manual

Copyright © 1987 Unisys Corporation
All Rights Reserved
Unisys is a trademark of Unisys Corporation.

Relative to Release
Level 7.2

Priced Item

August 1987
Distribution Code SD
Printed in U S America
5024789

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a Field Communication Form (FCF) with the CLASS specified as 2 (S.W.:System Software), and the Type specified as 1 (F.T.R.), and the product specified as the 7-digit form number of the manual (for example, 5024789).

TABLE OF CONTENTS

Section	Title	Page
	ABOUT THIS DOCUMENT	xvii
	PURPOSE	xvii
	SCOPE	xvii
	AUDIENCE	xvii
	PREREQUISITES	xvii
	HOW TO USE THIS DOCUMENT	xvii
	ORGANIZATION	xviii
	RESULTS	xix
	RELATED DOCUMENTS	xix
1	OVERVIEW	1-1
2	LANGUAGE CHARACTERISTICS	2-1
	GENERAL	2-1
	Notations	2-1
	Optional Words	2-1
	Key Words	2-1
	Lower Case Words	2-1
	Braces	2-1
	Brackets	2-2
	Consecutive Periods	2-2
	Period	2-2
	BASIC SYMBOLS	2-2
	RESERVED WORDS	2-4
	LANGUAGE STATEMENTS	2-5
	IDENTIFIERS	2-5
	Scope of Identifiers	2-6
	Duplicate Identifiers	2-6
	Special Identifiers	2-6
	ARRAYS	2-7
	SUBSCRIPTING	2-7
	LITERALS	2-10
	Numeric Literal	2-10
	Non-Numeric Literal	2-10
	Undigit Numeric Literals	2-10
	CONTROLLER FIELDS	2-11
	FORMAT OF BPL PROGRAMS	2-13
	Block Format	2-13
	Program Entry Point	2-14
	Program Size Considerations	2-14
3	STATEMENTS	3-1
	GENERAL	3-1
	DECLARATION STATEMENTS	3-1
	EXECUTABLE STATEMENTS	3-1
	PROCEDURE CALL Statement	3-1
	DO UNTIL Statement	3-2
	WHILE DO Statement	3-3
	IF Statement	3-3
	CASE Statement	3-4

TABLE OF CONTENTS

Section	Title	Page
	ASSIGNMENT Statements	3-4
	COMPILER DIRECTING STATEMENTS	3-4
4	DECLARATION STATEMENTS	4-1
	GENERAL	4-1
	ADDRESS	4-2
	BIT	4-4
	CDATE	4-5
	COMMON	4-6
	CONTROL	4-7
	DATA DECLARATION	4-9
	DEFINE	4-16
	DYNAMIC	4-19
	FILE	4-21
	LABEL	4-31
	PICTURE	4-32
	PROCEDURE	4-33
	SUBROUTINE	4-37
	UNSEGMENTED	4-38
5	EXECUTABLE STATEMENTS/CONTROL AND ASSIGNMENT	5-1
	GENERAL	5-1
	ACCEPT	5-2
	ACCUMULATOR CONSTRUCTS	5-3
	ARM	5-5
	ASSIGNMENT	5-7
	BREAKOUT	5-21
	CASE	5-22
	CLOSE	5-25
	COMMENT	5-29
	COMPARE	5-30
	COPY	5-31
	DISARM	5-32
	DISPLAY	5-33
	DO	5-34
	DOZE	5-38
	DUMP	5-39
	EDIT	5-40
	ENTER	5-41
	EXIT	5-42
	EXITBLOCK	5-43
	EXITCASE	5-44
	EXITCOND	5-45
	EXITLOOP	5-46
	EXITROUTINE	5-47
	FILL	5-48
	GO	5-49
	IF	5-50
	LOCK	5-54
	OPEN	5-55

TABLE OF CONTENTS

Section	Title	Page
	OVERLAY	5-57
	Procedure Call	5-58
	READ	5-59
	SCAN	5-61
	SEARCH	5-63
	SEARCH LINK/DELINK	5-65
	SEEK	5-68
	SORT	5-70
	SORT RETURN	5-72
	SPACE	5-73
	SPOMESSAGE	5-74
	STOP	5-75
	STOQUE	5-76
	STORE	5-79
	Subroutine Call	5-80
	TOPLOOP	5-81
	TRACE	5-82
	TRANSLATE	5-83
	UNLOCK	5-89
	WAIT	5-90
	WHILE	5-92
	WRITE	5-93
	ZIP	5-95
6	COMPILER DIRECTING STATEMENTS	6-1
	GENERAL	6-1
	Conditional Compiling	6-2
	@LIBR	6-4
	@PAGE	6-5
	@ICM Declaration	6-6
	IFF Conditional Compiling	6-7
7	DATA COMMUNICATIONS	7-1
	GENERAL	7-1
	ACCEPT	7-2
	CANCEL	7-3
	CONDCANCEL	7-4
	DISPLAY	7-5
	ENABLE	7-6
	FILL	7-7
	INTERROGATE	7-9
	READ	7-13
	READY	7-14
	TRANSTBL	7-15
	WAIT	7-16
	WRITE	7-17
	WRITEREAD	7-18
	WRITEREADTRANS	7-19
	WRITETRANSREAD	7-20

TABLE OF CONTENTS

Section	Title	Page
8	PORT FILES	8-1
	GENERAL	8-1
	CLOSE	8-2
	GET	8-3
	IF	8-4
	OPEN	8-5
	PORT	8-6
	READ	8-8
	SET	8-9
	WAIT	8-11
	WRITE	8-13
	PORT FILE ATTRIBUTES	8-14
	FUNCTION OUTPUT PARAMETERS	8-17
9	READER SORTER - PRE-4A CONTROL CONSTRUCTS	9-1
	GENERAL	9-1
	READER SORTER FILE HANDLING	9-1
	SPECIFIC STATEMENT FORMATS	9-1
	ACTION 0 (Pocket Select)	9-2
	ACTION 4 (Pocket Light)	9-3
	ACTION 6 (Batch Count)	9-4
	ACTION 8 (Delay)	9-5
	OPEN	9-6
	READ	9-7
10	READER SORTER - DLP/4A CONTROL CONSTRUCTS	10-1
	GENERAL	10-1
	READER SORTER FILE HANDLING	10-1
	SPECIFIC STATEMENT FORMATS	10-1
	ACTION 10 (Pocket Select)	10-2
	ACTION 11 (Pocket Light Generate)	10-3
	ACTION 12 (Status Inquiry)	10-4
	ACTION 13 (Charateristics Inquiry)	10-5
	ACTION 14 (Microfilm Advance)	10-6
	ACTION 15 (Start Flow)	10-7
	CLOSE	10-8
	OPEN	10-9
	READ	10-10
	BUFFER	10-11
	USER FILE STATEMENT	10-11
11	OPERATING INSTRUCTIONS	11-1
	GENERAL	11-1
	Compiler Operational References	11-6
	File Equate Information	11-7
	Input	11-7
	CANDE Editor Format Files	11-8
A	BPL RESERVED AND KEY WORDS	A-1

TABLE OF CONTENTS

Section	Title	Page
B	HOW TO WRITE A BPL PROGRAM	B-1
	GENERAL	B-1
	WRITING RULES	B-1
	FORM OF A BPL PROGRAM	B-1
	PROCEDURE CALLING	B-5
	Relationships	B-5
	TABLE CREATION	B-7
C	WARNING AND ERROR MESSAGES	C-1
	GENERAL	C-1
	WARNINGS	C-3
	ERRORS	C-6
D	INDEPENDENTLY COMPILED MODULES (ICM)	D-1
	TYPE I ICMs	D-2
	Parameters	D-2
	COMMON Blocks	D-6
	LINKAGE Construct	D-8
	FORTRAN ICM Considerations	D-12
	TYPE II AND TYPE III ICMs	D-13
	BPL Language Constructs for Type II and Type III ICMs	D-14
	MODULE NAME DECLARATION	D-15
	PROGRAM ENTRY POINT DECLARATION	D-16
	ENTRY DECLARATION	D-17
	EXTERNAL DECLARATION	D-18
	Programming Considerations for Type II and Type III ICM	D-20
	Example	D-21
	THE BPLBND PROGRAM BINDER	D-27
	Functional Description	D-27
	BPLBND Input Statements	D-27
	BPLBND INPUT SELECTION STATEMENTS:	D-30
	REQUIRED Statement	D-30
	OPTIONAL Statement	D-31
	BPLBND OPTION STATEMENTS	D-32
	FATAL Statement	D-32
	NOEXTEND Statement	D-33
	BPLBND PRINT STATEMENT	D-34
	PRINTALL Statement	D-35
	PRINTANALYSIS Statement	D-36
	PRINTCODE Statement	D-37
	PRINTSEGANALYSIS Statement	D-38
	PROGRAMLIMIT Statement	D-39
	PROGRAMSIZE Statement	D-40
	STACKSIZE Statement	D-41
	BPLBND SEGMENTATION STATEMENTS	D-42
	SEGMENT Statement	D-42
	OVERLAY Statement	D-44
	BPLBND TERMINATOR STATEMENT	D-46
	END STATEMENT	D-46
	Input-Output Facilities of BPLBND	D-47

TABLE OF CONTENTS

Section	Title	Page
	Debugging and Diagnostic Facilities of BPLBND	D-48
	CODE AND DATA INFORMATION, ADDRESSES AND REFERENCES	D-48
	PARAMETER CHECKING	D-48
	ERROR HANDLING	D-48
	Operational Considerations for BPLBND	D-55
	BPLBND Examples	D-56
	Example 2:	D-59
E	COMMON BPL PROGRAMMING ERRORS	E-1
F	EBCDIC, USASCII, AND BCL REFERENCE TABLE	F-1
	GENERAL	F-1
G	BPL68	G-1
INDEX		1

LIST OF ILLUSTRATIONS

Figure	Title	Page
4-1	Format of ADDRESS	4-2
4-2	Format of BIT	4-4
4-3	Format of CDATE	4-5
4-4	Format of COMMON	4-6
4-5	Format of CONTROL	4-7
4-6	Format of Data Declaration, Option 1	4-9
4-7	Data Declaration, Option 2	4-11
4-8	Data Declaration, Option 3	4-12
4-9	Data Declaration, Option 4	4-12
4-10	Format of DEFINE	4-16
4-11	Format of DYNAMIC	4-19
4-12	Declaration of DYNAMIC	4-20
4-13	Determining the Size of Memory	4-20
4-14	Format of LABEL Declaration	4-31
4-15	Format of PICTURE Declaration	4-32
4-16	Format of PROCEDURE Declaration	4-33
4-17	Format of SUBROUTINE Declaration	4-37
4-18	Format of UNSEGMENTED	4-38
5-1	Format of ACCEPT Statement	5-2
5-2	Format of ARM Statement	5-5
5-3	Format of the Assignment Statement, Option 1 (MOVE)	5-7
5-4	Format of the Assignment Statement, Option 2 (EXCHANGE)	5-9
5-5	Format of the Assignment Statement, Option 3 (MOVE DATA, CONTROL OP B4700 only)	5-9
5-6	Format of the Assignment Statement, Option 4 (COMPUTE)	5-10
5-7	Format of the Assignment Statement, Option 5 (LOGICAL OPERATORS or BOOLEAN OPERATORS)	5-12
5-8	Format of the Assignment Statement, Option 6 (SPECIAL BRANCH COMMUNICATES)	5-13
5-9	Format of the Assignment Statement, Option 7 (SEGDICT)	5-13
5-10	Format of the Assignment Statement, Option 8 (SEGMENT)	5-14
5-11	Format of the Assignment Statement, Option 9a (INTERROGATE FILE on disk)	5-14
5-12	Format of the Assignment Option 9b (INTERROGATE FILE on Diskpack)	5-14
5-13	Format of the Assignment Statement, Option 10a (PROGRAM PARAMETER BRANCH COMMUNICATES, ANY MCP)	5-15
5-14	Format of the Assignment Statement, Option 10b (PROGRAM PARAMETER BRANCH COMMUNICATES, PRE-MCP/VS 2.0)	5-15

LIST OF ILLUSTRATIONS

Figure	Title	Page
5-15	Format of the Assignment Statement, Option 10c (PROGRAM PARAMETER BRANCHCOMMUNICATES, MCP/VS 2.0 AND LATER)	5-15
5-16	The CASE Statement, Format 1	5-22
5-17	The CASE Statement, Format 2	5-22
5-18	Format of the CLOSE Statement	5-25
5-19	Format of the COMMENT Statement	5-29
5-20	Format of the COMPARE, Option 1	5-30
5-21	Format of the COMPARE, Option 2	5-30
5-22	Format of the COPY Statement	5-31
5-23	Format of the DISARM Statement	5-32
5-24	Format of DISPLAY, Option 1	5-33
5-25	Format of DISPLAY, Option 2	5-33
5-26	Format 1 for DO	5-34
5-27	Format 2 for DO	5-34
5-28	Format of the DOZE Statement	5-38
5-29	Format of the DUMP Statement	5-39
5-30	Format of the EDIT Statement	5-40
5-31	Format of the ENTER Statement	5-41
5-32	Format of the EXIT Statement	5-42
5-33	Format of the EXITBLOCK Statement	5-43
5-34	Format of the EXITCASE Statement	5-44
5-35	Format of the EXITCOND Statement	5-45
5-36	Format of the EXITLOOP Statement	5-46
5-37	Format of the EXITROUTINE Statement	5-47
5-38	Format of the FILL Statement	5-48
5-39	Format of the GO Statement	5-49
5-40	Format 1 for the IF Statement	5-50
5-41	Test of Condition-1 with IF, Option 1	5-50
5-42	Test for Condition-1 with IF, Option 2	5-51
5-43	Test for Condition-1 with IF, Option 3	5-52
5-44	Format 2 for the IF Statement	5-53
5-45	Format of LOCK Statement	5-54
5-46	Format of OPEN Statement	5-55
5-47	Format of OVERLAY Statement	5-57
5-48	Format for a Procedure Call	5-58
5-49	Format of READ Statement	5-59
5-50	Format of SCAN Statement	5-61
5-51	Format of the SEARCH Statement	5-63
5-52	Format of the SEARCH LINK DELINK Statement	5-65
5-53	Format of the SEEK Statement	5-68
5-54	Format of the SORT Statement	5-70
5-55	Format of the SORT RETURN Statement	5-72
5-56	Format of the SPACE Statement	5-73
5-57	Format of the SPOMESSAGE Statement	5-74
5-58	Format of the STOP Statement	5-75
5-59	Format for STOQUE Statements	5-76
5-60	Format of the STORE Statement	5-79
5-61	Format of a Subroutine Call	5-80

LIST OF ILLUSTRATIONS

Figure	Title	Page
5-62	Format of the TOPLOOP Statement	5-81
5-63	Format of the TRACE Statement	5-82
5-64	Format of the TRANSLATE Statement	5-83
5-65	Translate Tables in Memory	5-84
5-66	B Address (Identifier-1) Modification	5-84
5-67	B Address (Identifier-2) Modification	5-85
5-68	Format of the UNLOCK Statement	5-89
5-69	Format of the WAIT Statement	5-90
5-70	Format of the WRITE Statement	5-93
5-71	Format of the ZIP Statement	5-95
6-1	Format of the Conditional Compile, Double Dollar-Sign Record	6-2
6-2	Format of the @LIBR Statement	6-4
6-3	Format of the @PAGE Statement	6-5
6-4	Format of the ICM Declaration	6-6
6-5	Format of the IFF Statement	6-7
7-1	Format of DATACOMM ACCEPT	7-2
7-2	Format of DATACOMM CANCEL	7-3
7-3	Format of DATACOMM CONDCANCEL	7-4
7-4	Format of DATACOMM DISPLAY	7-5
7-5	Format of ENABLE	7-6
7-6	Format of FILL	7-7
7-7	Format of DATACOMM INTERROGATE	7-9
7-8	Format of READ	7-13
7-9	Format of READY	7-14
7-10	Format of TRANSTBL	7-15
7-11	Format of WAIT	7-16
7-12	Format of WRITE	7-17
7-13	Format of WRITEREAD	7-18
7-14	Format of WRITEREADTRANS	7-19
7-15	Format of WRITETRANSREAD	7-20
8-1	Format of the CLOSE Statement	8-2
8-2	Format of the GET Statement	8-3
8-3	Format of IF Interrogating Identifier-1	8-4
8-4	Format of the OPEN Statement	8-5
8-5	Format of the PORT Declaration	8-6
8-6	Format of the READ Statement	8-8
8-7	Format of the SET Statement	8-9
8-8	Format of the WAIT Statement	8-11
8-9	Format of the WRITE Statement	8-13
9-1	Format of ACTION 0	9-2
9-2	Format of ACTION 4	9-3
9-3	Format of ACTION 6	9-4
9-4	Format of ACTION 8	9-5
9-5	Format of OPEN	9-6
9-6	Format of READ	9-7
10-1	Format of ACTION 10	10-2
10-2	Format of ACTION 11	10-3
10-3	Format of ACTION 12	10-4
10-4	Format of ACTION 13	10-5

LIST OF ILLUSTRATIONS

Figure	Title	Page
10-5	Format of ACTION 14	10-6
10-6	Format of ACTION 15	10-7
10-7	Format of CLOSE	10-8
10-8	Format of OPEN	10-9
10-9	Format of READ	10-10
B-1	Compile Time Relationships of Procedures	B-5
B-2	The Scope of a Procedure	B-6
D-1	Format of the LINKAGE Declaration	D-8
D-2	First Source Record of an ICM2/3	D-15
D-3	Format of the PROG-ENTRY Declaration	D-16
D-4	Format of ENTRY	D-17
D-5	Format of the EXTERNAL Declaration	D-18
D-6	Type II ICM Example, First Module	D-23
D-7	Type II ICM Example, Second Module	D-25
D-8	Type II ICM Example, Third Module	D-26
D-9	Format of the REQUIRED Statement	D-30
D-10	Format of the OPTIONAL Statement	D-31
D-11	Format of the FATAL Statement	D-32
D-12	Format of the NOEXTEND Statement	D-33
D-13	General Format of PRINT Options	D-34
D-14	Format of PRINTALL	D-35
D-15	Format of PRINTANALYSIS	D-36
D-16	Format of PRINTCODE	D-37
D-17	Format of PRINTSEGANALYSIS	D-38
D-18	Format of PROGRAMLIMIT	D-39
D-19	Format of PROGRAMSIZE	D-40
D-20	Format of STACKSIZE	D-41
D-21	SEGMENT with Data Blocks	D-42
D-22	Definition of the OVERLAY Statement	D-44
D-23	Mapping of Overlayed Regions	D-45
D-24	Format of the END Statement	D-46
D-25	BPLBND Example, Control Statement Listing	D-60
D-26	BPLBND Example, Program Information Listing (Sheet 1 of 8)	D-60
D-26	BPLBND Example, Program Information Listing (Sheet 2 of 8)	D-61
D-26	BPLBND Example, Program Information Listing (Sheet 3 of 8)	D-62
D-26	BPLBND Example, Program Information Listing (Sheet 4 of 8)	D-63
D-26	BPLBND Example, Program Information Listing (Sheet 5 of 8)	D-64
D-26	BPLBND Example, Program Information Listing (Sheet 6 of 8)	D-65
D-26	BPLBND Example, Program Information Listing (Sheet 7 of 8)	D-66
D-26	BPLBND Example, Program Information Listing (Sheet 8 of 8)	D-67

LIST OF TABLES

Table	Title	Page
2-1	Characters That Define Conditions	2-3
2-2	Punctuation in BPL	2-4
2-3	BPL Statements	2-5
2-4	Special Identifiers	2-6
2-5	Examples of literals	2-10
2-6	Controller Field Reserved Words	2-11
4-1	Declaration Statements	4-1
4-1	CONTROL Options	4-7
4-2	Declaration Types and Sizes	4-11
4-3	Definitions	4-17
4-4	Allowable Hardware-Name Entries	4-23
4-5	Allowable Recording Modes	4-25
4-6	The Functions of TRANSLATE Values	4-27
4-7	Integer Settings in the I/O DESCRIPTOR	4-27
4-8	Allowable Routine Types	4-28
4-9	LABEL-use Routines	4-29
5-1	Assignment Overrides	5-8
5-2	Assignment Overrides in Arithmetic Operations	5-11
5-3	Names of Special Brand Communicate Instructions	5-13
5-4	Special Names for Use with Any Current MCP	5-16
5-5	Communicates	5-16
5-6	Special Names for Option 10c	5-17
5-7	Relational Operators	5-51
5-8	Permitted Logical Operators	5-52
5-9	Calling Procedures	5-58
5-10	READ Constructs	5-60
5-11	Shared Disk SEEK Constructs	5-69
5-12	Keyboard Commands in SPOMESSAGE	5-74
5-13	Storage Queue Parameter Block	5-77
5-14	Translate Table Address	5-86
7-1	Result Descriptor Digits	7-10
7-2	The Status of Result Descriptor Digits	7-11
8-1	Port Attributes	8-15
8-2	Subfile Attributes	8-16
9-1	Routine Types and Their Functions	9-1
10-1	Routine Types and Their Functions	10-1
11-1	Bits in the Value Statement	11-5
11-2	Internal and External I/O File Names	11-7
B-1	A Typical BPL Program	B-2
D-1	BPLBND Input and Files	D-47
D-2	Declaring the Five Segments	D-58

ABOUT THIS DOCUMENT

PURPOSE

This document is intended to describe the various features of the BPL programming language, and to provide reference material for programmers who make use of this language.

SCOPE

This document describes the BPL language. Specifically, this document describes both the programming language accepted by the BPL compiler and various options and control statements used with this compiler. This document is not intended as a teaching device but as a reference guide only.

AUDIENCE

The primary audience of this document includes experienced programmers who create programs using BPL or who need to understand programs previously written in BPL. A possible secondary audience can include programmers attempting to learn BPL, but the document is NOT structured for such an audience.

PREREQUISITES

This document is designed for the use of experienced programmers. Programmers using this document should be familiar with the general concepts and language-independent principles of programming.

HOW TO USE THIS DOCUMENT

To use this document for general understanding of BPL:

Read Sections 1 and 2

To find information about a particular BPL construct:

Locate the desired contruct in Section 3, 4, 5 and 6

To find information about the relationship between BPL and data communication:

Read Section 7 and 8

To find information about the special reader sorter constructs of BPL:

Read Section 9 and 10

To find specific information about the programming in BPL:

Read Section 11 and Appendices B and E

Other reference material is included throughout the document. Read Organization later in this section to find the location of other reference material.

ORGANIZATION

This document is divided into eleven sections. Seven appendices have also been provided. Brief descriptions of the sections and appendices are provided in the following paragraphs.

SECTION 1: BASIC OVERVIEW

This section contains a brief description of the concepts of BPL. In this short overview, the programmer can become familiar with concepts and terminology that are basic to the language.

SECTION 2: LANGUAGE CHARACTERISTICS This section presents detailed and specific information about the language characteristics of BPL. In general, the material present in this section varies from notations to the formatting of BPL programs.

SECTION 3: STATEMENTS

This section provides a general description of the three main classifications of statements: declaration, executable and compiler directing.

SECTION 4: DECLARATION STATEMENTS

This section examines declaration statements in detail. The beginning of this section lists these statements in alphabetical order.

SECTION 5: EXECUTABLE STATEMENTS - CONTROL AND ASSIGNMENT

This section examines executable statements in detail. These statements perform the data transformation and decision-making functions of a BPL program, and are described in alphabetical sequence in this section.

SECTION 6: COMPILER DIRECTING STATEMENTS

This section shows you how to use constructs for compiler directing statements. Examples and syntax statements are included for each type of statement such as: forms control, library routine functions, the building of Independently Compiled Module (ICM) and conditonal compiling.

SECTION 7: DATA COMMUNICATIONS

This section describes the BPL constructs required to activate the data communications equipment as defined by the FILE statement. Specific formats as well as detailed descriptions are presented for each construct.

SECTION 8: PORT FILES

This section contains detailed information as well as program examples and syntax statements concerning port files, which are a useful means of interprogram communication.

SECTION 9: READER SORTER - PRE-4A CONTROL CONSTRUCTS

This section describes the BPL constructs required to activate the READER SORTER equipment as defined by the FILE statement SORTER clause. Specific statement formats as well as detailed descriptions are presented for each construct.

SECTION 10: READER SORTER - DLP/4A CONTROL CONSTRUCTS

This section describes the BPL constructs required to activate the READER SORTER equipment connected to the system through a DLP (V Series and B 900-series systems) or through a 4A I/O Control.

SECTION 11: OPERATING INSTRUCTIONS

This section describes the procedures used to compile a BPL program.

APPENDIX A: BPL RESERVED AND KEY WORDS

This appendix contains a list of BPL reserved words in two categories: Class I (RESERVED) and Class II (KEY).

APPENDIX B: HOW TO WRITE A BPL PROGRAM

This appendix provides the necessary tools in writing a BPL program. It is assumed that the user has a general understanding of programming techniques.

APPENDIX C: WARNING AND ERROR MESSAGES

This appendix contains a list of warning and error numbers with their respective descriptions.

APPENDIX D: INDEPENDENTLY COMPILED MODULES (ICM)

This appendix describes how the BPL compiler can generate Independently Compiled Modules (ICMs). Descriptions are given of three types of ICMs: ICM1s, ICM2s and ICM3s. Also included is a detailed description of the BLPBND program binder and its activation with the ICM2s. For examples of ICM3s, see the B 2000/B 3000/B 4000/V Series BINDER Programming Reference Manual.

APPENDIX E: COMMON BPL PROGRAMMING ERRORS

This appendix describes some errors commonly made in writing BPL programs.

APPENDIX F: EBCDIC, USASCII, AND BCL REFERENCE TABLE

This appendix contains tables of the EBCDIC, USASCII and BCL character sets.

APPENDIX G: BPL68

This appendix provides a list of DEFINES which are specified in the BPL68 library file.

RESULTS

After using this doucument, the programmer should be more familiar with the notations and constructs of the BPL language.

The programmer should be able to find the answers to specific questions about the BPL language, and to interpret syntax in existing BPL programs.

RELATED DOCUMENTS

B 2000/B 3000/B 4000/V Series MCPIX System Software Operation Guide, Volumes 1 and 2, for MCPIX running on either B 2000/B 3000/B 4000 and V Series Systems.

V Series MCP/VS System Software Operation Guide, Volumes 1 and 2, for MCP/VS 1.0.

V Series MCP/VS System Software Operation Guide, Volumes 1 through 4, for MCP/VS 2.0 or greater.

B 2000/B 3000/B 4000/V Series MCP Programmers' Guide

BNA Architectural Description Reference Manual, Volume 1

B 2000/B 3000/B 4000/V Series BINDER Programming Reference Manual

SECTION 1
OVERVIEW

BPL is an ALGOL-like language which has been designed to make all hardware capabilities available at the machine language level, and to offer full flexibility in the specification of instructions and data. Constructs are provided for all MCP-program interfaces giving the BPL compiler most of the capabilities of an assembler, plus the advantages of a high-level language.

BPL employs a vocabulary of reserved words and symbols. The use of these reserved words and symbols to create a program is defined by the language description in this manual.

A BPL program has a distinct format which specifies the relative location of two major program categories: declarations and executable statements.

Declarations are provided in the language for giving the compiler information about the constituents of a program, such as array sizes, types of values that variables may assume, or the existence of procedures. Executable statements specify the functions or transformations to be performed upon the contents of storage.

The results produced by evaluation of arithmetic expressions can be assigned as the values of variables by means of assignment statements. These assignments are the principle active elements of the language. In addition, to provide control of the computational processes and external communication for a program, certain additional statements are defined to provide iterative mechanisms, conditional and unconditional program control transfers, and input/output operations. In order to provide control points for transfer operations, statements may be labeled.

Statements are composed of symbols which, in turn, are composed of letters, digits, and special characters. Symbol strings are then called operands, operators, or control functions. The BPL syntax is concerned with the legal creation of symbol strings and the relative placement of the strings to form executable or declarative statements.

A series of statements enclosed by the reserved words BEGIN and END is called a compound statement. If a declaration of identifiers appears immediately after the word BEGIN and prior to the related statements, the statement group is called a block. Both compound statements and blocks provide a method for grouping related statements, and they therefore can be the constituents of still more compound statements and blocks. A program is a grouping of such statements.

A program written in BPL, called a source program, is accepted as input by the BPL compiler. The compiler verifies that all rules outlined in this manual are satisfied, and translates the source program language into an object program language capable of communicating with the Master Control Program and directing the computer to operate on the desired data. Should source corrections become necessary, appropriate changes can be made and the program recompiled.

SECTION 2
LANGUAGE CHARACTERISTICS

GENERAL

Detailed and specific information about the characteristics of BPL is included on the following pages.

Notations

The notation convention that follows enables the reader to interpret the BPL syntax presented in this manual.

Optional Words

Optional words are included in BPL to improve the readability of the statement formats. All upper case words not underlined may be included or excluded from the source program. If they are included, they must be spelled correctly. For example, GO TO A ... is equivalent to GO A .... Therefore, the inclusion or omission of the word TO does not influence the logic of the statement.

Key Words

All underlined, upper case words are key words within a statement and are required when the functions of which they are a part are utilized. Their omission will cause syntax error conditions at compilation time.

For example:

IF {identifier} THEN statement [ELSE statement]

The key words are IF, THEN, and ELSE.

All underlined special characters shown in the syntax are key symbols and must be indicated in the position shown. For example, in a syntactical item such as *INTEGER (DYNAMIC)* .... the parentheses are required.

Lower Case Words

All lower case words represent generic terms which must be supplied in that format position by the programmer. "Identifier", "expression", and "statement" are generic terms in the preceding example.

Braces

When words or phrases are enclosed in braces { }, a choice of one of the entries must be made. With reference to the preceding gexample, one of the items (identifier or expression) must be included in the statement.

Brackets

Words and phrases enclosed in the brackets [ ] represent optional portions of a statement. If the programmer wishes to include the optional features, he may do so by including the entry as shown between the brackets; otherwise, it may be omitted. In terms of the preceding example, the ELSE statement may be included in the statement as an option.

Consecutive Periods

The presence of an ellipsis (...) within any format indicates that the syntax immediately preceding the notation may be successively repeated, depending upon the requirements of problem solving.

Period

The period, or dot, is used to override previously defined attributes of identifiers.

For example:

```
WORK.+2.2.UN
```

provides an override of any previously defined length or data type of the variable WORK.

BASIC SYMBOLS

The BPL character set is composed of:

- The upper and lower case letters A through Z.
- The digits 0 through 9.
- The break character – (underscore).
- The arithmetic operators + (addition), - (subtraction), * (multiplication) and / (division) to provide mathematical capabilities.
- The logical operators AND, OR, EOR, and NOT (negation).

  NOTE
  The logical operators may not always generate the same operation. See the ASSIGNMENT construct and the IF construct.

- The assignment symbol := (or replacement).
- The BPL Compiler accepts the following characters in conditional relations:

Table 2-1. Characters That Define Conditions

Notation	Meaning
= (or EQL)	equal
^= (or NEQ)	not equal
< (or LSS)	less than
<= (or LEQ)	less than or equal
> (or GTR)	greater than
>= (or GEQ)	greater than or equal
	(^ represents a logical "not" character)

The not sign (^), when used alone, is equivalent to a logical NOT.

The not sign (^) will not print correctly if printing on a BCL printer.

The double special characters must be written as shown above. For example, =< would be an illegal representation of less than or equal (<=).

- The following table defines the function of each punctuation symbol used in BPL:

Table 2-2. Punctuation in BPL

Symbol	Definition	Use
.	Period or dot	Attribute overrides
,	Comma	Item separator
:	Colon	Label delimiter
;	Semicolon	Statement delimiter
(	Left parenthesis	Enclose parameter lists
)	Right parenthesis	Enclose parameter lists
"	Quotation mark	Left and right character delimiter
#	Pound Sign	Right text string delimiter
	Space or blank	Data-name delimiter
<	Left arrow	Assignment or replacement
@	At sign	Enclosing undigit literals
[	Left bracket	Enclosing subscripts or denote address constants
]	Right bracket	Enclose subscripts or denote address constants
%	Percent sign	Enclose literals
:=	Colon-equal	Assignment or replacement
?	Question mark	In column 1 indicates an MCP Control record
$	Dollar sign	In Column 1 indicates a compiler control record
&	Ampersand	Remainder of card is a comment

RESERVED WORDS

There is within BPL a set of character strings, called RESERVED WORDS, with preassigned meanings. Two classes of Reserved Words are defined.

Class I words have preassigned meanings throughout an entire program. Some examples are: EXIT, PROCEDURE, DO, END. Incorrect usage of a Class I reserved word will always result in a syntax error.

Class II reserved words have preassigned meanings only within certain BPL statements. Examples are LINK, STACK, PARITY, ENABLE. Incorrect usage of a Class II word within a specific BPL statement results in a syntax error. The usage of a class II Reserved word in any other portion of the program is considered as a separate and distinct usage and will not result in a syntax error.

A full list of all classes of reserved words is provided in Appendix A.

LANGUAGE STATEMENTS

There are seven types of statements in BPL. Their names and a brief description of their functions are:

Table 2-3. BPL Statements

Type	Function
Declaration	Reserves space for, and assigns attributes to identifiers.
Executable	Performs data transformations and decision-making functions.
Control	Iterates, groups, or transfers control to sets of statements.
Procedure	Defines a subset of the program to be used as a subroutine.
Conditional	Controls the execution of individual statements or groups of statements.
Assignment	Performs calculations and/or assigns a value to an identifier (data-name).
Compiler Directing	Assists the programmer in preparing, formatting and compiling a program.

IDENTIFIERS

Identifiers are used to name labels, variables, arrays, procedures, files, and so forth. An identifier is created from a combination of not more than 30 characters, selected from the following:

A through Z,
a through z,
0 through 9,
The special character "underscore"

NOTE
Labels over 30 characters long will be truncated and warning 0205 will be generated.

An identifier must start with a letter, which can be followed by any combination of letters, digits, or both. The latter restriction also applies to labels, since integer labels are specifically disallowed. An identifier is terminated by a space, comma or semicolon. An identifier may not contain a special character (except underscore) or a space, and may not be a reserved word. An "underscore" may not begin an identifier.

Scope of Identifiers

Each block automatically introduces a new level of nomenclature; therefore, any named declaration occurring within the block is said to be local to the block in question. Such a declaration means:

- The entity represented by the identifier inside the block has no existence outside the block.
- Any entity represented by the same identifier outside the block is inaccessible inside the block.

An identifier occurring within an inner block and not declared within that block will be nonlocal (or global) to it; that is, the identifier will represent the same entity inside the block and in the level or levels immediately outside it, up to and including the level in which the identifier was declared.

Since a statement within a block may itself be a block, the concepts of local and nonlocal to a block must be understood. An identifier which is nonlocal to block A may or may not be nonlocal to block B in which block A is one statement.

A label must be declared in the head of the innermost block in which the associated labelled statement appears. If any statement in a block is labelled, the declaration of this label must appear within the block.

Duplicate Identifiers

There exists the possibility of having duplicate identifiers in BPL which do not cause a compile time error. This is true whenever the duplicate identifiers are declared in different blocks. Duplicate identifiers within one block are an error and will result in a syntax error.

Duplicate identifiers do not interfere only because they exist within the scope of their blocks. The case may occur, however, when the block wich contains the duplicate name is nested within the block that contains the first occurrence of the name. The compiler resolves this conflict by referencing the most recent occurrence of the name over the scope of the nested block. When this block returns control to the outer block, the original name is again available.

Special Identifiers

Four special identifiers are provided to facilitate memory management and indexing. Their names (reserved words) and attributes are as follows:

Table 2-4. Special Identifiers

Name	Location	Size	Type
BASE	0	1	UN
IX1	8	Signed 7	SN
IX2	16	Signed 7	SN
IX3	24	Signed 7	SN

These identifiers may be used on either side of an assignment statement. When using the index registers, caution should be shown regarding two areas.

- When using the index registers as unsigned numeric fields (UN), you must place a controller size of .8 to get the entire 8 digit field (normal size is 7), and a controller .UN to override its signed numeric properties.
- The values contained in the index registers may be changed by certain types of statements.

IX1 Case statements, Search statement, and Subscripting

IX2 Certain IO statements (FILE has IX2 ON), and SEARCH DELINK.

IX3 Procedure/Subroutine calls and EXITs.

It is the programmer's responsibility to store and restore any significant index values.

ARRAYS

An array is a repetitive set of data-elements. The identifier used with the array definition becomes the name of the entire array and individual elements in the array are addressed by subscripting. Arrays are single dimensional, that is, they allow only one value in the subscript. The ARRAY declaration may be used with all data-types except BIT.

SUBSCRIPTING

Within an array, the particular element is referenced by using subscripts. A subscript follows the identifier representing the array in a BPL statement, and must be contained in brackets. A space may separate the identifier and the subscript. A subscript may be either a numeric literal or an identifier. An identifier used as a subscript may not itself be subscripted.

At the point an identifier is used for subscripting purposes, its value must be greater than or equal to zero, but not greater than the value shown in the referenced ARRAY clause. The generated object code will not check the validity of values used for subscripting, and undefined results will occur should the program reference a subscripted identifier containing a negative value, or a value above the defined subscript range as reflected in the ARRAY clause pertaining to that item. The first entry of an ARRAY is always referenced with a subscript value of zero.

For example:

```
EXAMPLE1:
BEGIN
     INTEGER A(6);
     INTEGER ARRAY B [9] (6)
     INTEGER C (1);
          C:=5;
          A:=B[C];
        IX1:=30;
          A:=B[IX1]
END;
```

In the above example A is a six digit field, B is an array containing 10 six digit entrys. C is a one digit field used as a subscript. The statements A:=B[C] and A:=B[IX1] are equivalent in the above example, with the exception that A:=B[C] will generate an extra instruction to compute the value of C.

The following rules apply to subscripting:

- The first data name subscript variable will utilize IX1 regardless of other uses of IX1 in other subscript entries in the same statement.

  For example:

```
EXAMPLE 2:
BEGIN
     INTEGER ARRAY A [9] (6);
     INTEGER B (1);
     INTEGER ARRAY C [5] (6);
     INTEGER D (5):=5;
                 B:=4;
           A [B]:= C[IX1] +D;
END;
```

  Both A and C above will be subscripted by IX1. The above example is equivalent to:

```
A[4]:=C[4] + 5;
```

  The first data-name subscript variable will not utilize IX1 if IX1 has been used previously in the statement as other than a subscript variable.

  For example:

```
EXAMPLE 3:
BEGIN
      INTEGER A (10);
      INTEGER ARRAY B [5] (10)
      INTEGER ARRAY C [5] ( 4);
      INTEGER I (1);
      A. IX1:=B[I]+C[IX1];
END;
```

BPL will use indirect addressing, not IX1 to compute the address of B. Indirect address usage will cause an extra instruction to be generated.

The normal multiply will be generated using a temporary storage area. This is followed by an INC of the address of B into the temporary area which is then used as an indirect address in the addition of B to C.

- Subscripts may be signed, however, negative subscripting will cause undefined results.
- Checking for subscript values that exceed table size is the responsibility of the user.
- A negative value in an index regester, when that register is used as a subscript variable, will cause undefined results.
- Use of IX1, IX2, or IX3 as a subscript data name assumes the user has set the corresponding index register to the desired value. It should be noted that when a subscript variable uses an index register with controller overrides, it is considered a data name. Particular use of IX1 with controller overrides may cause undesirable results.

  For example:

```
EXAMPLE4:
BEGIN
      INTEGER A (6);
      INTEGER B (4);
      INTEGER ARRAY C [5] (6);
      IX1:=B;
      A:=C [ IX1.+4.4.UN ]
END;
```

  The designation C[IX1.+4.4.UN] is, in this case, effectively the same as C [B].

- A subscript variable is required when referencing an array name, wherever it is used. For example, if the "ADDRESS OF" an array name is needed, it is written:

  A := [arrayname [0]] ;

  when used within an address constant only a literal subscript is valid.

- If both data names in a subscripted statement have controller overrides the statement should appear as follows:

  A.4.+3.UN [B.3.UN] :=...

LITERALS

A literal is an item of data which contains a value identical to the characters being described. There are three classes of a literal: numeric, non-numeric, and undigit.

Numeric Literal

A numeric literal is defined as an item composed of characters chosen from the digits 0 through 9, the plus sign (+) or minus sign (−), and the decimal point.

- There must be at least one digit in a numeric literal.
- The sign of a numeric literal must appear as the left-most character. If no sign is present, the literal is defined as a positive value.
- The decimal point may appear anywhere within the literal except for the right-most or left-most character of a numeric literal. A decimal point within a numeric literal identifies the literal as a REAL number. Absence of a decimal point denotes an integer.
- A numeric literal used for arithmetic manipulations cannot exceed 99 signed digits.

Non-Numeric Literal

A non-numeric literal may be composed of any allowable character. The beginning and ending of a non-numeric literal is denoted by a quotation mark. Any character enclosed within quotation marks is part of the non-numeric literal. Subsequently, all spaces enclosed within the quotation marks are considered part of the literal. Two consecutive quotation marks within a non-numeric literal cause a single quote to be inserted into the literal string. Four consecutive quotation marks will result in a single " literal.

A non-numeric literal cannot itself exceed 99 characters.

Undigit Numeric Literals

Hexadecimal values 10 through 15 are represented as A through F, and must be bound by @ signs when used. For example, hexadecimal 11 would be literalized by @B@. A hexadecimal literal cannot exceed 99 digits. Hexadecimal values 10 through 15, when enclosed in percent signs (%), will represent numeric literals in byte format. For example, %F2% would cause a one-byte literal to be generated.

Table 2-5. Examples of literals

```
123          & numeric literal (integer)
1.49         & real number
@12@         & same as 12
@4F@         & two digit literal of 4F
"ABC"        & alpha literal
"AB""C"      & an alpha literal of AB"C
%C1C2C3%     & same as "ABC"
%4F%         & 1-byte alpha literal of 2 4-bit values of 4F
"4F"         & 2 alpha literals - alpha 4, alpha F (F4C6)
""""""       & a single alpha literal of a quotation mark
```

CONTROLLER FIELDS

Controller fields are used to override the natural attributes of the associated data name. Each controller field is free form and order is not important. Each controller field must be preceded by a period (.). The following is a list of valid controller field reserved words:

Table 2-6. Controller Field Reserved Words

UN	-	Unsigned numeric (4-bit)
UA	-	Unsigned alpha (8-bit)
SN	-	Signed numeric (4-bit)
IA	-	Indirect address (4-bit)
NM	-	8 bit numeric
NO	-	No hardware controller desired.Used with address constants and is also used to override the generation of extended addressing
IX1	-	Index register 1
IX2	-	Index register 2
IX3	-	Index register 3

NOTE

Indirect addressing and indexing may be used on most fields in statements that produce BCTs. Indirection must be only one address deep for correct compilation.

The following is a list of valid controller field overides:

Unsigned numeric literal

This number is considered an override field length and must be no greater than 6 digits long.

The override length will be in digits if the data item, after any attribute overrides, is of a 4-bit type. This would include, for example, an item declared INTEGER and not overridden to a UA or NM status, or any item with a controller override listed above as 4-bit.

The override length will be in bytes if the data item, after any attribute overrides, is of an 8-bit type. This would include, for example, an item declared ALPHA or NUMERIC, or any item with a controller override listed as 8-bit.

Signed numeric literal

Increment/decrement offset to the associated data name. This number may be up to 6 digits long. Multiple increment or decrement controllers are allowed on a single operand. The resultant offset on the operand is the sum of all increment or decrement operations.

Example: A.+4.−2.+1 is the same as A.+3.

Signed identifier

Plus or minus prior to an identifier takes the length of the identifier as a plus or minus offset to the associated data name.

Identifier

An identifier in a controller override can be either the name of an indirect field length area or the name of a data field whose length is used as a length override.

Indirect field length area:

If the identifier names a field located at address 38 or below, and the identifier is not preceded by a + sign, the contents of the field will be used as an indirect field length. That is, the value in that field will be taken as the length of the data name whose attribute is being overriden. If the field contains zero, a length of 100 will be used. (This is the only mechanism for obtaining variable field lengths.)

Length of field as length override:

If the identifier names a field whose address is above address 38, the length of that field is used as the length of the data name whose attribute is being overridden. (See EXAMPLE5, notes 5 and 6.)

Index register

The address of the data name is offset (incremented/decremented) by the value in the specified index register. The following limitations apply:

- The offset is always in digits regardless of other overrides.
- The sign of the value in the index register determines whether the offset is an increment or a decrement. A + or − sign may not precede the index register name.
- Only one index register at a time can be used in a data name's override.
- The index register can only be used as an address offset, not as a length override. REAL DOUBLE or FIXED operands can only have the indexing overrides specified.

```
EXAMPLE5:
BEGIN
     INTEGER INFL (2) = 38;
     ALPHA ABC (6):= "ABCDEF";
     INTEGER DEF (10):=1234567890;
     INTEGER G (6):= [ABC.NO];& note 1
             G:= ABC.UN.+10.3:& note 2
             G:= ABC.INFL.UA.IX1;& note 3
             G:= ABC.INFL;& note 4
             G:= ABC.G.UN;& note 5
             G:= ABC.G.+G.UN;& note 6
END;
```

Notes:

1. G would contain the address of ABC with NO address controller. The NO controller override specifies that an address controller value of zero is desired regardless of the definition of ABC.
2. G would contain 3 digits; 2 from the end of ABC and 1 from the beginning of DEF. G contains 000C61.
3. If INFL contains 02 and IX1 contains 2 then G would contain 000023.
4. If INFL contains 1 then G contains 000001.
5. G would contain C1C2C3 the digit equivalent of the first 3 bytes of ABC.
6. G would contain information at six digits past the information in note 5.

FORMAT OF BPL PROGRAMS

BPL programs are segmented into logical subdivisions called blocks. Each block begins with a BEGIN statement and terminates with an END statement. Blocks have a definite relationship to other blocks within a program, either side by side (disjoint) or subordinate (nested).

A block is disjoint from any other block if neither is a statement within the other; and a block is nested if it is wholly contained within another.

Block Format

A block is the basic structural element in BPL. Blocks have a rigid internal structure: first, all label and identifier declarations and procedures; second, all executable statements, which may or may not include nested blocks. The structure of all nested blocks must be exactly the same.

A block becomes a segment when a BEGIN is followed by a declaration.

For example:

```
BEGIN
      A:=B;
END;
```

is considered a block, but has no effect on the physical program structure.

```
BEGIN
      INTEGER A (6);
END;
```

This block causes a new segment and its resulting overlay mechanism to be generated at this location. Segmentation can be overridden; see the UNSEGMENTED declaration.

Program Entry Point

Execution of a BPL program starts at the first executable statement in the outermost block; that is, the statement which follows all nested procedures. (See also Appendix B, How to Write a BPL Program, and Appendix D, Type II ICMs, under Program Entry Point.)

Program Size Considerations

The first executable instruction in an overlayable segment cannot begin above address 300000.

A procedure entry and exit can occur at an address above 300000. In place of a simple EXIT, the compiler generates a branch to a routine in low memory. This routine moves the 6-digit return address from the stack into an extended address field, and then exits to that extended address. This avoids the problems (such as unintended indirect addressing or indexing) which would otherwise occur when the high-order digit of the address is 3 or greater.

SECTION 3
STATEMENTS

GENERAL

Statements are the BPL equivalent of natural language sentences. They contain a complete sequence of operations (one complete idea) which are logically separate from other similar sequences. Where an expression evaluation results in a numerical value, statement evaluation specifies functions or assignments for the values. For example, the expression A + B results in a numerical value, while the statement X :=A + B; (read X is replaced by A + B) assigns the value of the expression to the data-name X.

Statements are always terminated by a separator (;, ELSE, END, DO, UNTIL).

Statements fall into three main classifications: declaration, executable, and compiler directing.

DECLARATION STATEMENTS

DECLARATION statements relate memory space and data attributes to data-names and procedure locations.

EXECUTABLE STATEMENTS

EXECUTABLE statements are further broken down into control and assignment statements.

CONTROL statements determine the sequence in which statements are to be executed. They pass control to procedures, bind groups of statements together or conditionally specify which one of several statements is to be executed next.

PROCEDURE CALL Statement

The major control statement in BPL is the PROCEDURE CALLing statement. It consists of a procedure-name, followed by any parameters enclosed in parentheses and terminated by a semi-colon. For example, a procedure ABS requiring one parameter would be invoked by the statement:

```
ABS (VALU);
```

There are two considerations governing the use of procedure calling statements. First the called procedure must be within the scope of the calling procedure.

Second the called procedure will always return control to its calling procedure. To return control, the programmer should generally structure the program logic to "fall through" the last END statement in the procedure, although alternate means are available and will be described later. The immediately following executable statement in the calling procedure is executed when control is returned.

DO UNTIL Statement

Statements may be bound or grouped together by the DO, IF ... THEN ... ELSE, or CASE statements. The DO statement binds all following statements up to an UNTIL statement as if they were one statement.

For example:

```
EXAMPLE6:
BEGIN
     INTEGER X(5);
     INTEGER A(4);
     PROCEDURE ROUTINE (Y,B);
          INTEGER Y (5);
          INTEGER B (4);
               BEGIN
                    X:=Y*B;
               END;
DO
BEGIN
     X:=X+1;
     A:=1;
     ROUTINE (X,A);
END UNTIL X>5;
END;
```

A DO group is always executed at least once. The individual statements within the group may be any executable statements including imbedded DO statements.

WHILE DO Statement

The WHILE condition DO statement performs iterations of the statements within the group until the WHILE condition is met.

For example:

```
EXAMPLE7:
BEGIN
     INTEGER X(5);
     INTEGER A(4);
     PROCEDURE ROUTINE (Y,B);
          INTEGER Y (5);
          INTEGER B (4);
               BEGIN
                    X:=Y*B;
               END;
          PROCEDURE PRTN;
          BEGIN
               X:=X+1;
               ROUTINE (X,A);
               WHILE X<4 DO PRTN;
          END;
PRTN;
END;
```

IF Statement

The conditional-expression within the IF statement, when evaluated, designates which of two statements is to be executed.

For example:

```
EXAMPLE8:
BEGIN
     SIGNED INTEGER A (1);
     SIGNED INTEGER B (1);
     SIGNED INTEGER X (1):=5;
     BEGIN
          IF A GTR X THEN DO
               BEGIN
                    A:=A-1;
                    B:=B+1;
               END
          UNTIL A LSS X
          ELSE DO
               BEGIN
                    A:=A-1;
                    B:=B-1;
               END UNTIL A GTR X;
END;
```

After the chosen statement executes, control passes beyond the end of the IF statement.

CASE Statement

The CASE statement is an expanded form of the IF statement. The conditional expression evaluation chooses one statement from among all the following statements up to the END statement for execution. After that one statement is executed, control passes to the first statement beyond the END statement.

DO, IF, CASE, PROCEDURE invocations or ASSIGNMENT statements may be imbedded in any of the above statements in any order and to any depth.

ASSIGNMENT Statements

The ASSIGNMENT operation moves the contents of one identifier, called the source field, into the memory-space of another identifier, called the destination field. Alignment, truncation or padding is performed during assignment and is controlled by the length attributes of the identifiers involved, and by the type of the receiving field.

The type attribute divides alignment control into two cases. The first case is an alphabetic move, which aligns the data-names on their left-most or high order characters. The assignment is then performed in a left to right order until one of the fields is exhausted. If the destination field is the shorter, data is truncated from the right. If the source field is the shorter, then the destination field is padded on the right with space characters (%40%).

The second case is a numeric move, where the receiving field is aligned on the right-most, or low-order digit. If the destination field is shorter than the number of significant digits in the sending field, the overflow indicator is set and the operation terminates. If the source field is the shorter, the destination field is left-filled with zeros.

COMPILER DIRECTING STATEMENTS

Compiler Directing statements include those which handle forms control, library routine functions, the building of Independently Compiled Modules (ICM), and conditional compiling.

SECTION 4
DECLARATION STATEMENTS

GENERAL

Declaration statements are detailed in this section. For fast reference, an alphabetical list of these statements is shown in table 4-1. This table lists the heading (in this section) where you will find a description of each declaration statement.

Table 4-1. Declaration Statements

Declaration	Heading
ADDRESS	ADDRESS
ALPHA	Data Declaration
ARRAY	Data Declaration
BIT	BIT
CDATE	CDATE
COMMON	COMMON
CONTROL	CONTROL
DEFINE	DEFINE
DOUBLE	Data Declaration
DYNAMIC	DYNAMIC
FILE	FILE
FIXED DOUBLE	Data Declaration
FIXED INTEGER	Data Declaration
FIXED REAL	Data Declaration
INDIRECT	Data Declaration
INTEGER	Data Declaration
LABEL	LABEL
MOD	ADDRESS/Data Declaration
NUMERIC	Data Declaration
OWN	Data Declaration
PICTURE	PICTURE
PROCEDURE	PROCEDURE
REAL	Data Declaration
SIGNED INTEGER	Data Declaration
SUBROUTINE	SUBROUTINE

ADDRESS

The ADDRESS declaration is used to reset the location counter at compile time to a predetermined address or memory location.

The format of the address is:

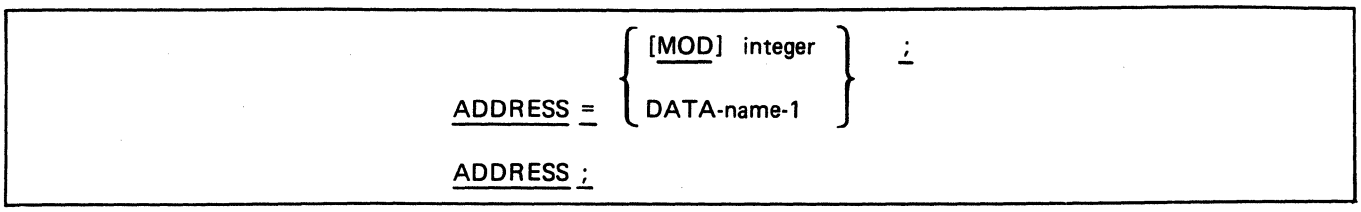


Figure 4-1. Format of ADDRESS

MOD, if used, must be followed by an integer which does not exceed four digits in length and which has a value greater than zero.

The use of MOD with a value other than 1, 2, or 4 will cause the entire block in which it appears to start at a MOD location that is the common denominator of all MOD statements within the block.

The "ADDRESS =..." construct may be used inside a PROCEDURE to redefine either stack relative or OWN local variables.

When ADDRESS is used with a register name (e.g., IX2), BASE or a literal, the address reverts to a BASE relative address. Otherwise, this construct is used to provide a segment relative address. It should be noted that data may be initialized while in a "segment relative" mode; but cannot be initialized while in a "base relative" mode at a location greater than the beginning of the segment dictionary, or 200, whichever is smaller.

"ADDRESS ;" resets the location counter to the point referenced prior to the last "ADDRESS =" statement.

It should be noted that "ADDRESS =" and "ADDRESS ;" are nested in a similar way to BEGIN/ENDS. Each "ADDRESS =" except "ADDRESS MOD m" must have a matching "ADDRESS ;" or a syntax error occurs.

Example 1:

```
BEGIN
      ADDRESS = BASE.+280;
      INTEGER A(10):=3;
END;
```

This is an illegal statement because an address greater than 200 is being initialized.

Example 2:

```
BEGIN
        INTEGER B(16) :=4;
        ADDRESS = B ;
        INTEGER C(16) :=5;
        ADDRESS;
END;
```

This statement is legal; segment relative location B and C will contain 0000000000000005.

Example 3:

```
BEGIN
        INTEGER A(1) = BASE;
        ADDRESS = A ;
END;
```

This is equivalent to:

```
BEGIN
        INTEGER A(1) = BASE ;
        ADDRESS = BASE ;
END;
```

BIT

The BIT construct is used to symbolically reference a bit in memory.

The format of the BIT declaration is:

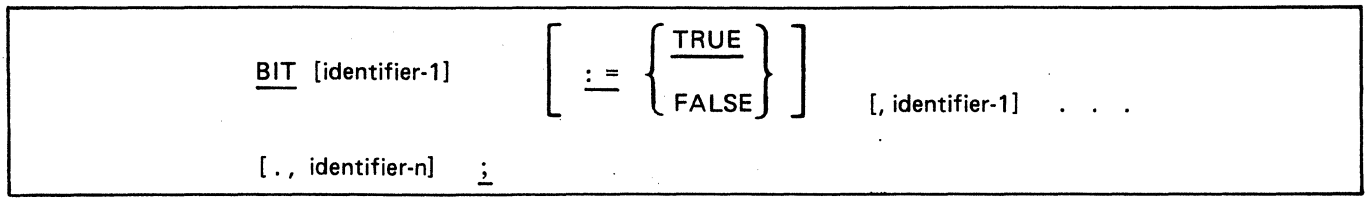


Figure 4-2. Format of BIT

Identifier-2 through identifier-n, specifying additional bits, must be separated by commas (,). If a semicolon (;) is used, the declaration is terminated.

If the value TRUE is specified, the bit will be set (on). If the value FALSE is specified, or if a value is not present, the bit will be reset (off).

For example:

```
EXAMPLE9:
BEGIN
        BIT A, B, C ;
        BIT D :=TRUE, E := FALSE ;
        BIT F ;
END;
```

Bits of a digit are allocated in the order: 8-bit, 4-bit, 2-bit, and 1-bit.

CDATE

The function of CDATE is to specify a compile time generation of the date compiled. The format for CDATE is:

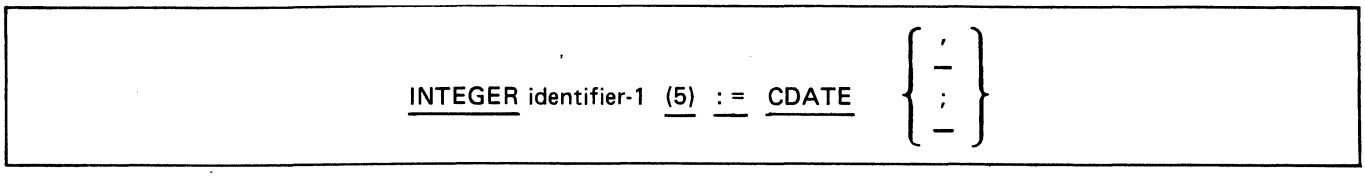


Figure 4-3. Format of CDATE

The compiler will store in identifier-1 the current date in Julian form (YYDDD).

CDATE may only be used in this form of declaration because it only has meaning at compile time.

COMMON

COMMON declarations are used to define and optionally initialize data areas being accessed by multiple ICMs.

The format of the declaration is:

```
COMMON identifier-1 BEGIN common-declarations ; END ;
```

Figure 4-4. Format of COMMON

Identifier-1 is a 6 character unique name by which the data area is known. Up to 100 uniquely named common blocks are permitted. Common declarations may only be used in an ICM.

Common declarations provide for the definition of type and length of data areas, and for the initialization of the data declared. Refer to the heading DATA DECLARATIONS in this section for the appropriate formats.

Example:

```
COMMON A BEGIN
         SIGNED INTEGER C(11), D(11); (See note 1.)
         END;

COMMON B BEGIN
         INTEGER E (4) := 1; (See note 2.)
         ALPHA JUNK (2); (See note 3.)
         SIGNED INTEGER F(11) := +3; (See note 4.)
         END;
```

Notes:

1. A 24 digit area has been set up consisting of two signed 11 digit fields.
2. E is preset to 1.
3. JUNK is an unitialized 2 character alpha field.
4. F is preset to C000000000003.

Refer also to Appendix D.

CONTROL

The CONTROL statement is used to provide the BPL compiler information regarding the hardware features and program format desired for the resultant object program.

The format of the CONTROL declaration is:

```
CONTROL option [, option . . . ] ;
```

Figure 4-5. Format of CONTROL

One or more options can be selected from the following list. When multiple options are specified, they must be separated by commas.

Table 4-1. CONTROL Options

[MEMORY] := integer-1	EXTENDED
STACK := integer-2	DICTIONARY := integer-3
OP {B3500 / B4700}	BREAKOUT {TAPE / DISK}

CONTROL MEMORY requires an integer-1 being no greater than 6 digits in length. The object program memory size will be set to the specified size, rounded up to the next modulo 1000 digits. Absence of this control will create an object program whose size is the size of the object program plus stack, rounded up to the next modulo 1000 digits.

CONTROL STACK requires an integer-2 being no greater than 6 digits in length. The object program stack size will be set to the specified length. Absence of this control will force a stack size of 1000 digits.

CONTROL OP specifies the valid instruction set. Absence of this statement will cause the B2500/B3500 OP code set to be considered valid. CONTROL OP B4700 permits the generation of certain machine instructions which were not available on B2500/B3500 processors, including accumulator operations, bit set and reset, and search linked list (SLL and SLD) instruction.

If set, CONTROL OP 4700 will remain in effect until reset by a CONTROL OP B3500 statement, and conversely. This allows B3500/B4700 programs to be maintained in a single symbolic file. Warning 0501 will be generated if an attempt is made to set (reset) an already set (reset) option.

Setting a data-name to zero with CONTROL OP 4700 will always produce a BIT RESET instruction. However, signed fields will not use the bit reset instruction, as this destroys the sign (@C@).

If CONTROL OP B4700 is not set, FIXED REAL, FIXED DOUBLE, and FIXED INTEGER declarations are not syntaxed. The following attributes are assigned in such a case:

```
FIXED REAL A ; - REAL A(8);
FIXED DOUBLE B ; - REAL B(16);
FIXED INTEGER C ; - SIGNED INTEGER C(7).
```

Warning 0500 will be produced when this situation exists. Note that mixed INTEGER and FIXED operand arithmetics and assignments cannot be handled without the OP B4700 option set.

CONTROL EXTENDED specifies that the program may exceed 100000 digits of data and/or coding, and that the extended address feature is available. All addresses within any segment that exceeds the above limit will have extended addressing, and all "INDIRECT" declarations and name parameters will have extended addressing. It is important that this particular control statement appear before any declaration that is affected by the above rules. If an arithmetic operation is performed on a variable declared INDIRECT and CONTROL EXTENDED is set, then the computation will be performed on the rightmost six digits (as if a ".+2.6" modifier was used).

If CONTROL EXTENDED is set and a segment exceeds 100KD (100000) digits, base relative and stack relative addresses will not be made extended if the program is not an ICM.

CONTROL DICTIONARY specifies (by integer-3) the base-relative address of the start of the segment dictionary for the program. If no declaration appears, the segment dictionary will begin at base-relative address 64. Integer-3 must be greater than or equal to 64, a MOD 4 address, and less than 6 digits in length. The programmer should note that base-relative data may be pre-initialized only up to base-relative address 200 or the start of the segment dictionary, whichever is lower.

CONTROL BREAKOUT is used to specify whether a programmatic BREAKOUT should be directed to magnetic tape or disk. If omitted, any BREAKOUT will be to system default.

NOTE

A CONTROL declaration should be made only once for each type of variation. CONTROL EXTENDED, when used, must be declared before any INDIRECT declarations. See examples under data declaration.

DATA DECLARATION

Data declarations are uued to define and optionally locate and/or initialize data areas within a BPL program.

The format of the data declaration statements are:

Option 1:

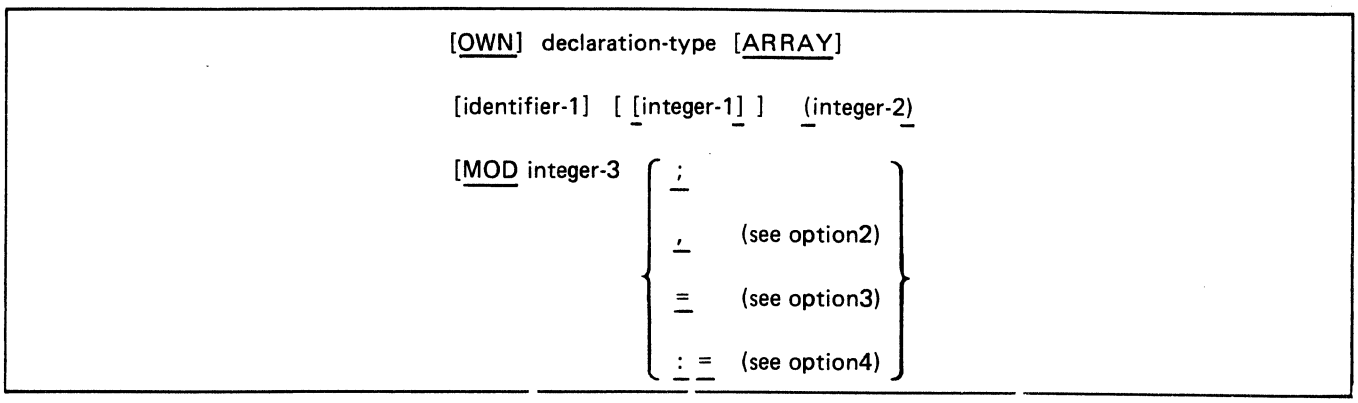


Figure 4-6. Format of Data Declaration, Option 1

The optional OWN declaration, if used within a procedure, causes those local variables preceded by OWN to become segment relative as opposed to stack relative. OWN placed before any data type outside of a procedure is ignored. The resources required for those declarations preceded by OWN are commanded only when that segment is in memory. This option offers to the programmer, a more efficient utilization of resources.

The declaration-type must be one of the following:

ALPHA
: to specify 8-bit alphabetic data. ALPHA type identifiers may not be used in an arithmetic operation unless a controller override is used (see CONTROLLER FIELDS).

BIT
: to define an individual bit as data which can be referenced symbolically. This declaration may contain a controller override, an index, or an increment or decrement. The order of allocation is 8-bit, 4-bit, 2-bit, 1-bit.

DOUBLE
: Use to define a real number for use in a double precision operation. Accumulator instructions will never be generated for any field defined in this category.

FIXED DOUBLE
: Use to define a real number having a sixteen digit mantissa for use in a double precision operation. If OP B4700 (see CONTROL statement) has been specified, accumulator commands will be used on this field whenever applicable.

FIXED INTEGER
Use to define an integer of seven digits plus sign. If OP B4700 (see CONTROL statement) has been specified, accumulator commands will be used on this field whenever applicable.

FIXED REAL
Use to define a real number having an eight digit mantissa. If OP B4700 (see CONTROL statement) has been specified, accumulator instructions will be used on this field whenever applicable.

INDIRECT
Use to define a field to be used to store an address. Although the INTEGER statement could be used for this purpose, INDIRECT will compile as a six or eight digit address depending on the CONTROL EXTENDED setting and will be forced to a MOD 2 address.

INTEGER
Use to define a numeric (4-bit) field.

NUMERIC
Use to specify a numeric field of 8-bit characters. Data defined as NUMERIC may be used in arithmetic operation.

REAL
Use to define a real number for use in a single precision operation. Accumulator instructions will never be generated for fields defined in this category. REAL and DOUBLE mean the same thing for B3500 floating operators.

SIGNED INTEGER
Use to define a signed numeric (4-bit) field.

The ARRAY option is used to define a sequence of data-items which possess identical formats. If the ARRAY option is used, any reference to identifier-1 must be subscripted (refer to SUBSCRIPTING). It is the user's responsibility to assure that the value of the subscript does not exceed the bounds of the ARRAY.

Identifier-1 is the name to be assigned to this memory area (with the specified data attributes). If it is not necessary to reference this declaration in the program, the identifier may be omitted.

Integer-1 specifies one less than the number of elements in an ARRAY and must be present when ARRAY is used. When the identifier associated with an ARRAY is referenced, subscripting must be used. If for example, integer-1 has a value of 5, the items in the ARRAY must be referenced as identifier-1 [0] through identifier-1 [5].

Integer-2 denotes the size of the entry, and must be enclosed in parentheses. This size takes on the attributes of the declaration-type; that is, the number of bytes for ALPHA, and the number of digits for INTEGER. A SIGNED INTEGER declaration will take one more digit of memory than specified for integer-2, this digit being the sign digit. REAL and DOUBLE will take 4 more digits than specified (sign, 2 digit exponent, sign). Integer-2 is required for all data declarations except those listed below, for which it is invalid.

Table 4-2. Declaration Types and Sizes

Declaration-Type	Size
FIXED DOUBLE	19 digits plus sign (16 digit mantissa)
FIXED INTEGER	7 digits plus sign (8.UN if OP 4700 is set)
FIXED REAL	11 digits plus sign (12.UN if OP 4700 is set)
INDIRECT	6 digits unless CONTROL EXTENDED is specified, in which case it is 8 digits.

MOD forces the address to the next exact multiple of a specified number (integer-3) unless the address is currently modulo integer-3. ALPHA, INDIRECT and NUMERIC are assumed MOD 2 unless otherwise specified. FIXED, DOUBLE, FIXED REAL, and FIXED INTEGER are always assumed MOD 4.

Reals cannot be mixed with other data types. Only FIXED operands and integers with mod-4 addresses and proper sizes (7SN, 8UN, 11SN, 12UN, 19SN or 20UN) that are not name parameters can be mixed within expressions, and only when CONTROL OP B4700 is set.

A semicolon is used to terminate the declaration.

Option 2:

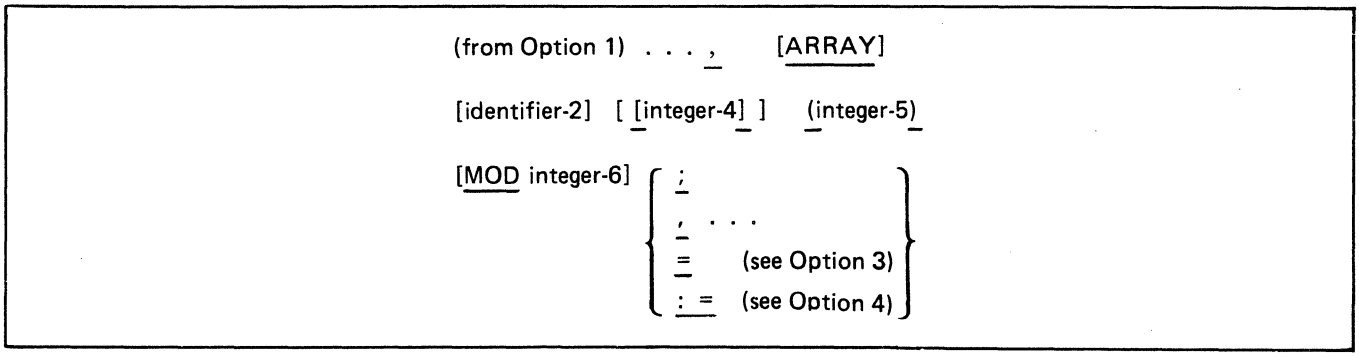


Figure 4-7. Data Declaration, Option 2

A declaration-terminator of a comma allows additional declaration of the same declaration-type. All fields following the comma have the same function as described for Option 1.

Option 3

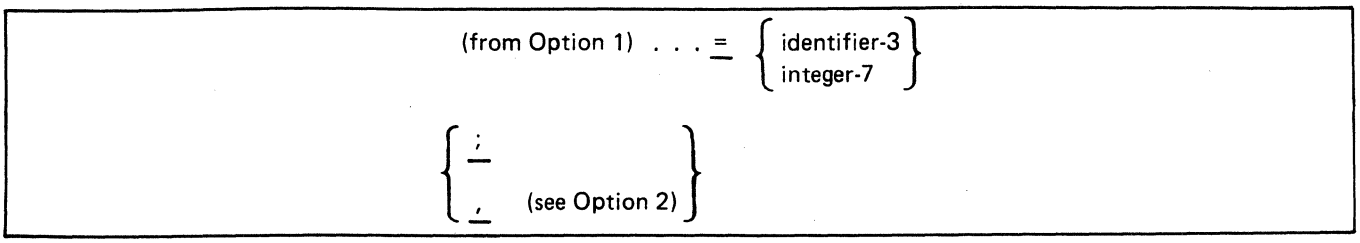


Figure 4-8. Data Declaration, Option 3

An equal sign is used to equate the identifier (identifier-1) address to the address of a previously defined identifier (identifier-3) or to a base relative address (integer-7).

Option 4

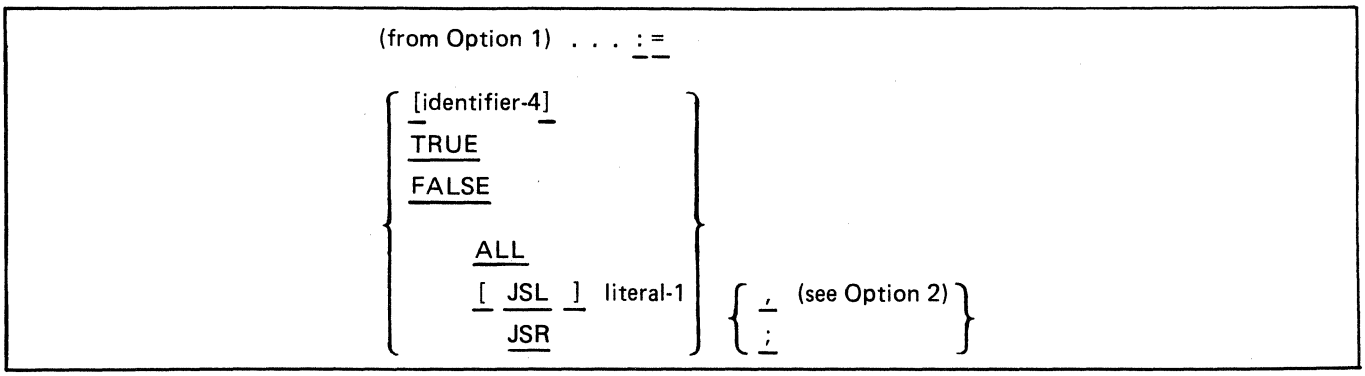


Figure 4-9. Data Declaration, Option 4

The colon-equal is used to preset the contents of the data area to either the ADDRESS OF identifier-4 or to a literal.

If identifier-4 is used, it must be enclosed in brackets. The contents of the data area will then be preset to the ADDRESS OF identifier-4.

If TRUE is specified, the data area identifier-1 will contain the value 1; if FALSE is specified, the value will be 0.

The ALL construct will force the entire data area to be initialized with repetition of the specified literal (literal-1).

JSL is used to left-justify INTEGER or NUMERIC fields, appending any trailing zeros required; and JSR is used to right-justify ALPHA fields, creating leading blanks when required.

If presetting an ARRAY, each element must be assigned a value.

NOTE
Uninitialized declarations will contain unpredictable data.

Examples:

EXAMPLE 10:

```
BEGIN
        CONTROL EXTENDED;                                   & See note 1
        CONTROL MEMORY:=200000;                             & See note 2
        INTEGER ARRAY BIG [999](100);                       & See note 3
        INTEGER A(5);                                       & See note 4
        ALPHA B(6),&(1):=".";                               & See note 5
        INTEGER C(6):=1;                                    & See note 6
        INTEGER D(8):=[A];                                  & See note 7
        INTEGER ARRAY E [3] (2):=01,04,08,13;               & See note 8
        INTEGER INFL38 (2)=38;                              & See note 9
        ALPHA F(200):=[ALL] "AB";                           & See note 10
        ALPHA G(5):=[JSR]"A";                               & See note 11
END;
```

Notes:

1. CONTROL EXTENDED is used to indicate that the program may exceed 100000 digits and that extended addressing will be generated whenever applicable.
2. This statement specifies that the program will be at least 200000 digits in size.
3. This entry will set up an array of 1000 entries of 100 digits each.
4. A is an uninitialized five digit field.
5. B is a six byte field, followed by a one byte field containing a period.
6. C is a six digit field preset to 000001.
7. D will contain the address of A in its extended form because it resides at an address over 100000 and CONTROL EXTENDED has been specified.
8. E is an array with four, two digit entries. These four entries are preset to 01, 04, 08, and 13 respectively.
9. INFL38 is a two digit field as absolute memory location 38, which may be used in indirect field length moves.
10. F will be preset by the compiler to ABABAB...
11. G will be preset to "bbbbA" (b = space).

EXAMPLE 11:

```
BEGIN
        CONTROL OP B4700;                          & See note 12
        FIXED REAL R;                              & See note 13
        FIXED INTEGER C:=1;                        & See note 14
        FIXED REAL D:=100.0;                       & See note 15
        FIXED DOUBLE D1:=100.0;                    & See note 16
        SIGNED INTEGER E(7):= +3;                  & See note 17
        SIGNED INTEGER F(7):= -0000003;            & See note 18
        INTEGER G(6):= "ABC";                      & See note 19
        ALPHA H(3):= 123;                          & See note 20
        INTEGER I(10):= [BASE.UA];                 & See note 21
        ALPHA J(3):= 12345;                        & See note 22
END;
```

Notes:

12. The statement specifies that a program be generated which will utilize machine instructions implemented since the B3500.
13. R is an uninitialized real number.
14. C is preset to 1.
15. D is preset to C03C10000000.
16. D1 is a double precision number preset to C03C1000000000000000.
17. E will be preset to C0000003.
18. F will be preset to D0000003.
19. G will contain 000123.
20. H will contain F1F2F3.
21. I will contain 0000200000.
22. J will contain F1F2F3 truncated.

Examples:

EXAMPLE 12:

```
BEGIN
        CONTROL STACK:= 500,
                BREAKOUT,
                DICTIONARY:= 200;
        ALPHA A(4) MOD 100:=12;                 & See note 23
        INDIRECT NAMEIA;                        & See note 24
        INDIRECT B:= [A];                       & See note 25
        INDIRECT ARRAY C [3]:                   & See note 26
        INDIRECT G:= [A.UA];                    & See note 27
        INDIRECT H:= [A.IX1];                   & See note 28
        INDIRECT I = A.+6;                      & See note 29
        NUMERIC J(4):= 1234;                    & See note 30
        DOUBLE K(12):= 100.0;                   & See note 31
        REAL L(8):= 100.0;                      & See note 32
END;
```

Notes:

23. A is at the next even 100 address available and is preset to F1F24040.
24. NAMEIA is the name of a field which may be used for indirect data declaration.
25. B contains the address of A.
26. This is an array of indirect address containers
27. G contains the address of A with an Alpha controller.
28. H contains the address of A with an IX1 controller.
29. I points to the address of A plus 6.
30. J contains F1F2F3F4.
31. K is preset to C03C100000000000.
32. L is preset to C03C10000000.

DEFINE

The DEFINE statement provides the capability to insert multiple copies of specified BPL source text into a program during compilation, from only one image of the source text; and to define often-used routines, with or without parameters, for use throughout a program.

Defines are transparent declarations that replace the calling DEFINE name with the defined portion. The DEFINE declaration assigns the meaning of the defined identifiers. Any reference causes the replacement of the defined identifier being referenced by the exact text, including all punctuation, which is associated with the identifier.

The format of the DEFINE declaration is:

```
DEFINE ident-1  [ (param. -1, param. -2, . . . ) ]  =  [defined-portion]  # ;
```

Figure 4-10. Format of DEFINE

Identifier-1 is required and when (and wherever) used in the BPL program will reference the specified definition (defined portion).

The parameter string is optional. If used, the DEFINE is known as a PARAMETRIC DEFINE. When a PARAMETRIC DEFINE is referenced an argument may or may not be included for each parameter specified. If excess arguments are provided, they will be ignored. If insufficient arguments are provided, the corresponding rightmost parameters will be dropped from the symbolic code.

The defined-portion may be any BPL language element except a nested DEFINE declaration. The defined-portion is optional; however, if omitted, spaces will replace the define call (identifier-1).

The number sign (#) is required to terminate the DEFINE, and the semicolon (;) is required to terminate the statement.

During compilation, syntax errors (if any) in a definition are noted following the use of the defined identifier.

A DEFINE statement must appear within the declaration section of the program or of a block. The scope of a DEFINE is the same as the scope of any identifiers in that declaration section; that is, it exists in its declaring block and all directly nested blocks.

Multiple DEFINEs may appear within one DEFINE statement and must be separated by commas.

Reserved words may be DEFINEd (used as identifier-1), but their special significance is lost within the scope of the DEFINE statement.

The actual parameters associated with an occurrence of a definition name are not restricted to simple identifiers. They may contain complex constructs but must be delimited by zero level commas, i.e., commas not enclosed within paired parentheses or braces. The actual parameters replace the formal parameters in the DEFINE statement in a left to right order and their number must be equal. The maximum number of parameters is limited to ten per definition-name.

Definitions can be nested, but not more than eight levels; that is, defined identifiers may be used in other definitions. For instance, in the table below, the definition for D3 is equivalent to the definition for DD. In the example, the definition AA is considered nested one level in the first declaration. In the second declaration, the definition AA is considered nested two levels, and so forth.

Table 4-3. Definitions

Examples	Comments
1. `DEFINE REPEAT = ABC (TAGA, X) #;`	The source code contained between the = and the # sign of the DEFINE statement will be copied into the BPL program whenever the word REPEAT is used.
`IF X EQL 9 THEN REPEAT;`	This statement is equivalent to IF X EQL 9 THEN ABC (TAGA, X);
2. `DEFINE IN = INTEGER # AL = ALPHA #; IN X (5); AL Y (4); IN Z (2);`	The source code generated would be: INTEGER X (5); ALPHA Y (4); INTEGER Z (2);
3. `DEFINE TRAIL (A, B, C) = IF A EQL ZERO THEN A := B ELSE C #; TRAIL (TAGA, ABS [BX], CX := SQRF [BX]);`	This statement generates the following: IF TAGA EQL ZERO THEN TAGA := ABS [BX] ELSE CX := SQRF [BX];
4. `DEFINE X = ABC #, ABC = X #;`	This statement will cause an diagnostic error at compile time when the compiler attempts to expand either X or ABC into its TEXT.
5. `DEFINE D1 = AA #; DEFINE D2 = D1 D1 #; DEFINE D3 = D2 D2 #; DEFINE DD = AA AA AA AA #;`	Nesting example. D3 and DD both generate the same symbolic code (AA AA AA AA).

Examples:

EXAMPLE 13:

```
BEGIN
INTEGER TAGA (5);
INTEGER X (1);
PROCEDURE ABC (A,B);
     BEGIN
     INTEGER A (5);
     INTEGER B (1);
     OWN INTEGER C (2);
          C := 4;
          A := A/C
          A := A*B;
     END;
DEFINE REPEAT = ABC (TAGA,X)#;
A: = 76;
B: = 2;
REPEAT;
REPEAT;
END;
```

EXAMPLE14:

```
BEGIN
DEFINE IN = INTEGER #, AL = ALPHA#;
IN X (5);
AL Y (4);
IN Z (2);
DEFINE X = ABC #, ABC = X#;     & ERROR NUMBER 2719 IF
DEFINE D1 = AA#;                & ABC OR X EVER USED
DEFINE D2 = D1; D1 #;
DEFINE D4 = D2; D2 #;
DEFINE DD = AA;AA;AA;AA#;
PROCEDURE AA;
     BEGIN
          IX1 := IX1 + 1;
     END;
D1;
D2;
D4;
DD;
END;
```

DYNAMIC

The DYNAMIC declaration provides a means of addressing the area of memory between the largest nested block (of the block in which the declaration is made) and the program stack.

The format of the DYNAMIC declaration is:

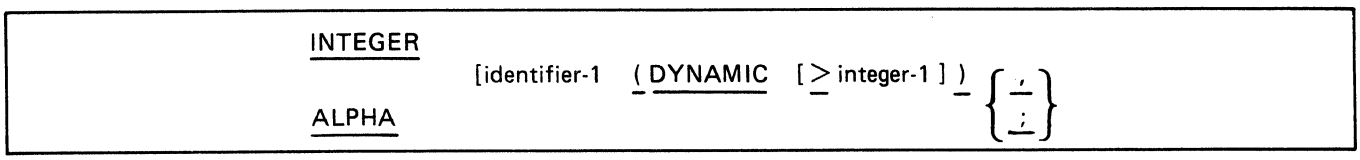


Figure 4-11. Format of DYNAMIC

The DYNAMIC declaration type must be ALPHA or INTEGER.

Identifier-1 is optional. If present, it is the symbolic name by which the DYNAMIC area is referenced; if absent, the DYNAMIC area may not be referenced symbolically.

Integer-1 specifies the minimum size of requested DYNAMIC areas and must be a numeric literal no more than six (6) digits in length. The size is expressed in units of bytes for ALPHA declarations and in units of digits for INTEGER declarations. A DYNAMIC area will always be adjusted to a MOD 4 address.

ADDRESS = MOD n within a dynamic block will reset the location counter for that block to a modulo-n value. The dynamic block for that segment will begin at an address that is the least common multiple of all mod factors for that block.

The use of a comma (,) allows more declarations of the same type (ALPHA or INTEGER) to follow; a semicolon terminates the DYNAMIC declaration. However, these declarations will not address the DYNAMIC area.

The DYNAMIC statement does not reserve memory space, nor does it affect the memory size of a program. If the requested (integer-1) amount of DYNAMIC memory is not available, a warning message will be provided at compile time. To increase the amount of available DYNAMIC space, a CONTROL MEMORY instruction must be provided.

Data areas in the DYNAMIC area may be defined by equating a data-name to the DYNAMIC name, or by an ADDRESS construct, however a DYNAMIC area cannot be pre-initialized. BPL does not check subscript range, therefore an array could be declared following an ADDRESS where the number of elements in the array is effectively ignored. Since the DYNAMIC area size may vary from compile to compile, the number of array elements would also vary.

It is the responsibility of the programmer to manage this DYNAMIC area (from end of coding to base of stack) using indexing since DYNAMIC may cross the 100 KD limit.

To use this feature, the user/programmer must do the following:

- Insert "DYNAMIC DECLARATIONS" in those segments where all subsections of coding are to be protected.

Example:

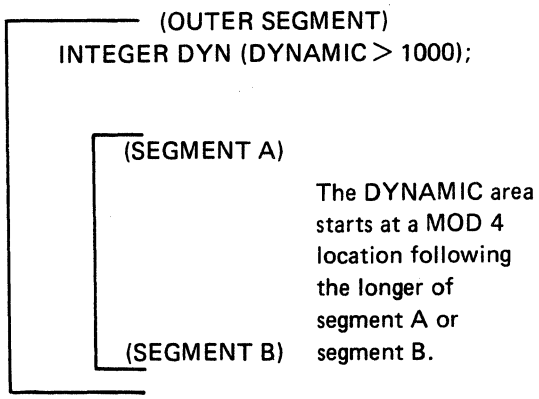

(OUTER SEGMENT)
INTEGER DYN (DYNAMIC > 1000);
(SEGMENT A)
The DYNAMIC area starts at a MOD 4 location following the longer of segment A or segment B.
(SEGMENT B)

Figure 4-12. Declaration of DYNAMIC

- Programmatically determine memory size prior to using the program stack. If memory size is greater than compiled size, add an adjustment factor to the stack pointer (BASE.+40.UN.6).

Example:

```
INTEGER TOTAL_MEM (6),
        STACK_WANTED (4)   := 2000;

TOTAL_MEM   := MEMORY;

BASE.+40.6   := TOTAL_MEM - STACK_WANTED;
```

Figure 4-13. Determining the Size of Memory

- Manage the DYNAMIC area by:
  - Determining its size by subtracting the address of its base from the program stack pointer or
  - Saving the original stack pointer and considering it to be a limit address for DYNAMIC.

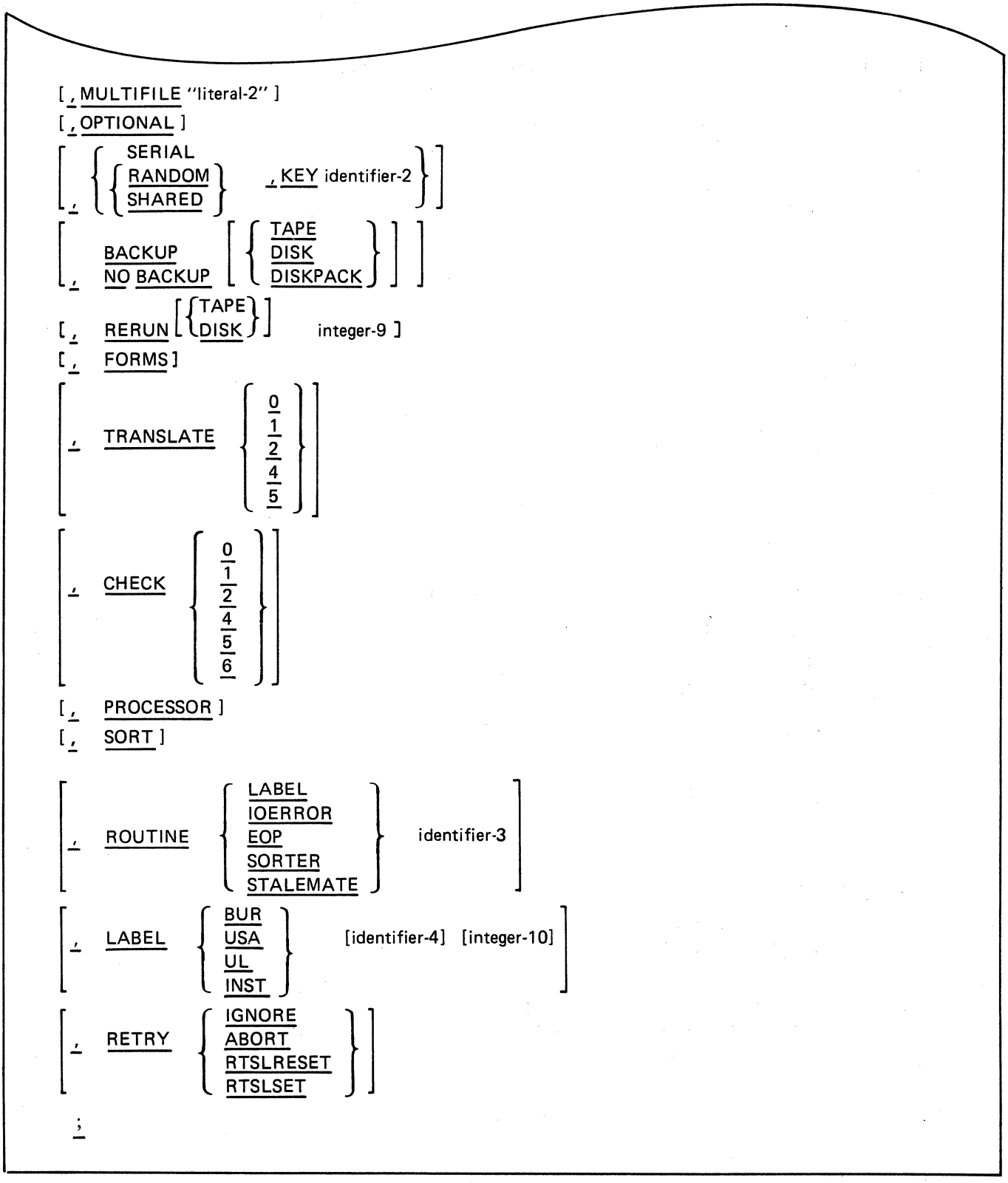
WARNING
If the stack pointer is adjusted following any procedure entry, then results are unpredictable.

FILE

The FILE declaration is used to establish file attributes.

The format of the FILE declaration is:

```
FILE file-name, hardware-name

[SINGLE]

[  [integer-1 BY] integer-2]

[, "literal-1" ]

, RECORD identifier-1   [ { ALPHA   } ]   integer-3
                         { INTEGER }

[, BUFFERS integer-4  [ = file-name-2]

[, BLOCKED integer-5]

[ , IX2  { ON  } ]
         { OFF }

[ , { WORKAREA    } ]
    { NO WORKAREA }

[, WORK]

[, READ AFTER WRITE]

[ , ASSIGN BY   { AREA     } ]
                { CYLINDER }
                { FILE     }
                { NN       }

[, SAVE integer-7]

[, SAVE FILE]

[ , MODE   { ALPHA   } ]
           { BINARY  }
           { EBCDIC  }
           { ODDPAR  }

[, VARIABLE [integer-8]]
```


[, MULTIFILE "literal-2"]
[, OPTIONAL]
SERIAL
RANDOM
SHARED
, KEY identifier-2
BACKUP
NO BACKUP
TAPE
DISK
DISKPACK
RERUN
TAPE
DISK
integer-9]
[, FORMS]
, TRANSLATE
0
1
2
4
5
, CHECK
0
1
2
4
5
6
[, PROCESSOR]
[, SORT]
, ROUTINE
LABEL
IOERROR
EOP
SORTER
STALEMATE
identifier-3
, LABEL
BUR
USA
UL
INST
[identifier-4] [integer-10]
, RETRY
IGNORE
ABORT
RTSLRESET
RTSLSET
;

The keyword FILE must be the first word in the FILE statement. The remaining entries may be in any order.

File-name is used to identify this file for all file handling statements (READ, WRITE, OPEN, CLOSE, etc.). File-name must be unique in the first six characters if the use of an MCP label equation (FILE) command is anticipated.

The hardware-name entry is required and defines the peripheral to be used by this file. Allowable hardware-name entries are:

Table 4-4. Allowable Hardware-Name Entries

B500	DISKPACK	TAPE (any type)
B774	DISPLAYUNIT	TAPEGCR (Group-coded recording tape only)
B2500	PRINTER	TAPEPE (Phase-encoded tape only)
B3500	PTPUNCH	TAPE7 (7-channel tape only)
B4700	PTREADER	TAPE9 (9-channel tape only)
B9350	PUNCH	TC500
B9352	READER	TC700
DCP	SORTER (Reader/Sorter not on DLP or 4A Control)	TOUCHTONE*
DISC	SORTER4	TWX
DISK	(Reader/Sorter on DLP or 4A Control)	

* TOUCHTONE is a registered trademark of A.T. & T.

The SINGLE option is for DISKPACK files only and is used to restrict a file to one pack. If omitted, a data file may be assigned to multiple packs.

The integer-1 BY integer-2 clause is required for DISK and DISKPACK files, and is invalid for any other devices. Integer-1 specifies the number of areas (pages) to be assigned to the file. If the integer is omitted, a default value of 20 will be used. The maximum number of areas allowed is 100.

Integer-2 specifies the number of records per area, and must be a multiple of the blocking factor. Although this clause is required for all DISK and DISKPACK files, the specified values will be ignored for those files OPENed INPUT.

Literal-1 (enclosed in quotes) is used to specify the external file-ID. If not present, the first six characters of the internal file name (file-name) will be used.

The RECORD entry is required and must be followed by an identifier (identifier-1) which specifies the RECORD name (and work area) for this file. Identifier-1 must be previously declared unless the NO WORKAREA technique is used, in which case the buffer address will be used when reference is made to the identifier. The ALPHA or INTEGER entry is optional, and is used to describe the data type of integer-3. If omitted, ALPHA will be used. Integer-3 is required and defines the maximum record size.

The BUFFERS clause is optional. If omitted, or if hardware-name is SORTER4 the default value of 1 is used. The maximum number of buffers (integer-4) which may be declared is 9, except that files of type SORTER or SORTER4 must have no more than 3 buffers. Each buffer requires additional memory space in the compiled object program. The file-name-2 option is used to equate the file buffers of the present file to the buffer area of the previously declared file-name(s). The original buffer area ((8 + buffer length) x number of buffers) must equal or exceed the buffer area size for the present file. Note that only buffer areas are being equated.

The function of the BLOCKED clause is to specify the number of logical records to be contained in a block. If this clause is omitted, records are assumed to be unblocked.

The IX2 option is used to set or reset the IX2 flag in the File Information Block. If set, each READ or WRITE performed for this file will cause the MCP to update IX2 to point to the beginning of the next logical record. This is primarily intended for use on blocked files using the buffer access technique (NO WORKAREA).

The default values are:

ON If NO WORKAREA and multiple buffers are used.

ON If NO WORKAREA and blocked records, regardless of number of buffers.

OFF If a work area is specified.

OFF If unblocked, one buffer, and NO WORKAREA is specified.

The NO WORKAREA option is used to specify that the records are accessed from the buffer area. A separate work area will not be assigned by the compiler. WORKAREA may be explicitly specified, if desired, for documentation purposes.

The WORK option specifies to the MCP that this DISK or DISKPACK file is to be used as a work file, and that the MCP should insert the program mix number in the second and third character position of the file-ID, thus creating a unique file-name at object run time. The use of this option allows multiprogramming of the same program without creating duplicate file-IDs for commonly used work files. If WORK is specified, PROCESSOR is assumed.

The READ AFTER WRITE option may be used after a DISK write to perform a read for parity check.

The ASSIGN clause is optional, and is permitted for DISK or DISKPACK files only. The allowable options are:

AREA
: For 100-byte DISK files, this option will assign areas to successive EUs or IDs within the default subsystem. For disk pack files, all areas after the first will be assigned to successive packs in such a manner that no pack with the same restriction status (as the pack with the first area) will have two areas of the file while another similarly restricted pack has none.

CYLINDER
: This option applies only to disk pack files, and specifies that areas be assigned by cylinder boundary. Cylinder boundary allocation can be limited in that the maximum areas assigned thus on a pack will be one less than the number of cylinders.

FILE
: This option applies only to 100-byte DISK files and assigns disk space on the default subsystem by disk file number (within the program).

NN
: This option applies only to 100-byte DISK files and assigns disk space on the EU, ID, or subsystem defined by NN. If subsystem assignment (as opposed to specific EU or ID) is desired, NN may be specified as 90 (default subsystem), or 91 - 93 (subsystem 1-3).

The SAVE option is used to specify the number of days a magnetic tape file is to be saved before it can be automatically purged by the MCP and used for other purposes. Integer-7 is limited to numbers 001 through 999. If the SAVE clause is omitted, a save factor of 1 is assigned to preclude expiration action when the system is being operated just prior to and shortly after midnight (2400).

The SAVE FILE option causes the file to be CLOSEd with LOCK by the MCP if the file is OPEN at End-Of-Job. (If a disk or diskpack file has not been previously saved in the directory, and SAVE FILE has not been specified, the file is purged if the creating program terminates without closing the file with RELEASE or LOCK.)

The MODE clause is used to specify the recording mode for certain peripheral devices. Standard recording mode is assumed if this clause is omitted. Allowable recording modes are:

Table 4-5. Allowable Recording Modes

DEVICE	STANDARD	NON-STANDARD
TAPE7	Odd Parity	Even Parity
PUNCH	EBCDIC	BCL
PTPUNCH	BCL	Binary

The word VARIABLE specifies a magnetic-tape file containing variable-length records. The user must specify the actual size (in bytes) of the variable-length records in the first four bytes of each record, and each record size must be an even number of bytes. The four-character variable-size indicator is included in the physical size of each record. Integer-8 must be used if the file is input to the SORT intrinsic to indicate the most frequently used record size.

The MULTIFILE clause is used with multi-file tape and disk pack files. Literal-2 is required immediately following the word MULTIFILE, and is used as the MULTIFILE-ID.

The word OPTIONAL is used to declare an INPUT file which may not be required to execute the job. If the file is not present at file OPEN time, the system operator may use the OF keyboard response to indicate an OPTIONAL file, thus forcing an end-of-file condition on the first READ of the file.

The DISK or DISKPACK file types may be used to specify as RANDOM, SHARED, or SERIAL. The default is SERIAL if this clause is omitted. If RANDOM is specified, the KEY clause is required. Identifier-2 has different meanings depending upon the hardware-type used. For DISK or DISKPACK, it is the identifier of the 8-digit actual key. For a SORTER file it contains the identifier of the manual end of file. For an OLBANKING file it contains the address into which the MCP stores the terminal unit number (DIGIT) upon completion of each I/O operation. SHARED indicates a DISK or DISKPACK file which may be shared between multiple processors. Files declared SHARED are assumed RANDOM and the KEY clause must be used. The BACKUP option will cause printer or punch files to be assigned to backup media depending upon the MCP options. BACKUP TAPE causes printer files to be assigned to printer backup tape, regardless of MCP option settings, and BACKUP DISK or DISKPACK causes printer or punch files to be placed on backup disk or diskpack, regardless of MCP options.

The NO BACKUP option will prevent the file from going to backup media unless otherwise specifically directed by the operator through an ODT message or label equate action.

The RERUN clause sets up a communication with the MCP to create periodic control points (breakouts) so that an operational program encountering a malfunction can be restarted at the last RERUN control point instead of restarting from the beginning of the program. Integer-9, the rerun counter cannot exceed five digits. TAPE or DISK may be indicated in the CONTROL statement to specify where the RERUN (breakout) information should be stored. (Program breakout is not supported on MCP/VS 2.0 and later.)

Use of the FORMS option with a PRINTER or PUNCH file will cause the MCP to halt the program at file OPEN time and to display a console message stating that special forms are required. For files sent to a backup medium (by the BACKUP option or an operator's action), the special forms message is displayed when the file is printed/punched.

The TRANSLATE option is used if code translation is to be performed on data before it is input to the program's buffer or written to the output medium.

The TRANSLATE values have the following meanings:

Table 4-6. The Functions of TRANSLATE Values

TRANSLATE Value	Function
0 (default for non-datacomm devices)	Most devices - no translation PTPUNCH, PTREADER - process 7-bit odd parity
1	PTPUNCH, PTREADER - translate BCL/EBCDIC (6-bit odd parity)
2	PTPUNCH, PTREADER - process 8-bit, no parity Datacomm - translate lower case to upper case (non-standard translation)
4 (default for datacomm devices)	Datacomm - Standard translation MT7 - translate BCL/EBCDIC
5	TAPEPE, TAPEGCR - translate ASCII/EBCDIC

The programmer must also be aware of the TRANSLATE executable statement, which allows digit/character translation by table.

The CHECK clause is for MICR files only (except those using 4A controls) and is used to specify which MICR check control is to be set in the I/O DESCRIPTOR. The allowable options are:

Table 4-7. Integer Settings in the I/O DESCRIPTOR

Integer	Function
0	Read and check all fields or 7.75 inches maximum.
1	End read validity check at seconds S2.
2	End Read at second S2.
4	Format and report errors in amount and transmit fields.
5	Both 1 and 4.
6	Both 2 and 4.

The PROCESSOR option causes the MCP to put the processor number in the fifth position of disk file file-IDs. Processor is assumed if the WORK option is specified.

The SORT option indicates to the compiler that the attributes of this file (record length, blocking factor, etc.) are to be used to compute the minimum amount of memory required for the SORT intrinsic.

The ROUTINE clause is used to specify procedures which are in addition to the standard procedures supplied by the MCP. These are commonly known as "use" routines,in reference to the COBOL convention of file USE routines in the DECLARATIVES section. BPL USE routines correspond to COBOL USE routines. Allowable ROUTINE types are:

Table 4-8. Allowable Routine Types

ROUTINE Type	Function
LABEL	Magnetic tape - identifies a label handling routine. SORTER - identifies the routine to process memory access, cannot read, unencoded, and double document errors.
IOERROR	Magnetic tape - identifies the routine to which the MCP will transfer control if it encounters an irrecoverable parity error on this file. Datacomm - specifies BREAK key procedure. SORTER - amount field error procedure.
EOP	PRINTER - end of page routine (channel 12). SORTER - transmit field error routine.
SORTER	SORTER - item pocket - select routine.
STALEMATE	DISK - contention use routine.Entered when MCP detects a stalemate condition on a shared file.

Identifier-3 is the procedure name and must immediately follow the ROUTINE type. In addition, identifier-3 must have been previously declared as a label.

A use routine is defined to be a cluster of program instructions, identified by a use routine label and terminated by the reserved word "EXITROUTINE" followed by the file name.

A use routine label is simply a BPL label occurring somewhere in the segment in which the file is declared. A use routine is not a procedure and must not be declared with a PROCEDURE declaration. It is merely a branch point within the mainline code of the segment containing the FILE declaration.

Since the use routine is a labeled routine, the normal blocking rules of BPL will guarantee that the use routine itself is not segmented outside of the segment containing the File Information Block. Each use routine has a special exit to the MCP and therefore should not be executed by other routines unless the EXITROUTINE communicate is bypassed.

A use routine is entered when the MCP detects the occurrence of the condition specified in the preceding paragraphs under FUNCTION, during an input or output operation possibly including OPEN/CLOSE on the file. It is then the programmer's responsibility to take any necessary action.

A LABEL routine is entered whenever a label is encountered or written on the file for which routine LABEL is specified. LABEL use routines have access to the following information through the reserved word ROUTINETYPE:

Table 4-9. LABEL-use Routines

Identifier	Indicator
ROUTINETYPE.1	0=Input, 1=Output.
ROUTINETYPE.+1.1	0=Beginning, 1=Ending.
ROUTINETYPE.+2.1	0=File,1=Reel.
ROUTINETYPE.+3.3	Reel number for multi-reel files

It is the programmer's responsibility to check ROUTINETYPE fields for the applicable label type.

The LABEL type clause is used to specify the type of label required for this file. Allowable types are: BUR to indicate Unisys standard label, USA to indicate USASCII standard label, UL to indicate an unlabelled file or INST to indicate an installation-defined label. Identifier-4 provides a symbolic reference to the label area and must not be previously declared. integer-10, when used denotes the length in bytes of any user area above the standard label size. For example, a Unisys standard label is 80 bytes in length. If integer-10 is used as a value of 20, the total label area reserved will be 100 bytes.

The RETRY option allows retry short/long (RTSL) action for variable length files. IGNORE initiates no special action, no matter what the setting of the MCPVI option RTSL is. ABORT always gives action on short/long reads. RTSLSET gives action only if RTSL is set (default).

The STALEMATE clause is used to specify the action to be taken when shared disk is used, and two processors are accessing the same file. While one processor may try to read a record which is locked by a second processor, the second proessor may try to read a record locked by the first processor. This condition will cause both processors to wait indefinitely unless the USE ON STALEMATE option is used.

Example:

```
BEGIN
     LABEL  LABELPROC;
     .
     .
     FILE X ..., ROUTINE LABEL LABELPROC, ...;
     .
     .
     & Procedure declarations
     .
     .
     & Executable code
     .
     .
     LABELPROC:
     & Program branches to this point when a label is
     &     detected on file X
     .
     .
     EXITROUTINE;
     .
     .
END;
```

LABEL

The LABEL declaration is used to reserve identifiers which will be used as control points within a procedure or block.

The format of the LABEL declaration is:

```
LABEL label-1  [ , label-2 , . . . , label-n ] ;
```

Figure 4-14. Format of LABEL Declaration

All LABEL declarations must appear in the declaration portion of the block in which they are to be used. Duplicate labels may appear in a program in different blocks; if duplicate labels appear within the same block, they cause a compile time syntax error.

When labels are used within the program body, they must be followed by a colon (:). They may only occur within coding. This is used to specify a control point which may be used in a transfer of control statement such as GO TO or end-of-file.

PICTURE

The PICTURE declaration is used with the EDIT statement to create an edited field, primarily for printing.

The format of the PICTURE declaration is:

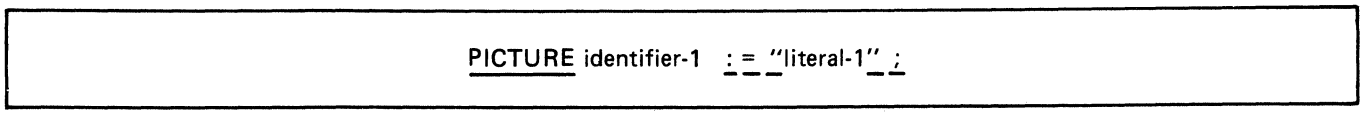


Figure 4-15. Format of PICTURE Declaration

The word PICTURE is required to identify the PICTURE declaration.

Identifier-1 may be any BPL identifier and will be used in the EDIT statement to reference the PICTURE declaration.

The colon-equal (:=) is required.

Literal-1 may be any valid COBOL editing picture, and must be contained in quotation marks and terminated with a semicolon.

The micro-operator string generated by the PICTURE declaration assumes a standard edit table at BASE.+48 containing its characters plus sign (+), minus sign (-), asterisk (*), period (.), comma (,), dollar sign ($), zero, and space.

Examples:

```
PICTURE PICT1 := "99/99/99";

PICTURE PIC2 := "Z(8)";

PICTURE PC3 := "ZZ9=99=9999";
```

PROCEDURE

A PROCEDURE declaration defines the procedure-identifier as the name of a procedure. Whenever the identifier followed by the appropriate parameters appears in the program, it produces a call upon the procedure.

The format of the PROCEDURE declaration is:

```
[FORWARD] [SEGMENTED] [level-number]  PROCEDURE  [.]

   procedure-identifier [(formal-parameter-list)] ;

  [VALUE value-parameter-list ;] [parameter-specifications ;]

  BEGIN [procedure-body-declarations ;] procedure-body

  END ;
```

Figure 4-16. Format of PROCEDURE Declaration

The optional word FORWARD indicates that the following procedure declaration is a FORWARD declaration and that the actual declaration follows elsewhere in the same block. FORWARD PROCEDURE declarations are used in order to satisfy a BPL rule that procedures must be declared before they are used. If A calls B and B calls A, one of them must be declared FORWARD with the actual declaration appearing later in the segment. FORWARD PROCEDURE declaration up to and including the parameter-specifications.

The optional word SEGMENTED is used to indicate that the following procedure is to be SEGMENTED (i.e., an overlay).

Level-number is used as a method to structure PROCEDURE overlay levels without having the PROCEDUREs physically structured. The level-number may be 0-99 inclusive with 0 being the default. Level numbers are relative to each other, within a block.

For example:

Program Declarations

```
PROCEDURE A;
SEGMENTED PROCEDURE B;
SEGMENTED 7 PROCEDURE C;
SEGMENTED 5 PROCEDURE D;
SEGMENTED PROCEDURE E;
PROCEDURE F;
SEGMENTED 7 PROCEDURE G;
```

Memory Layout

```
[A GLOBAL
[F GLOBAL

B [E  LEVEL 0

  [D      LEVEL 1 (5)

    [C  [G  LEVEL 2 (7)
```

The word PROCEDURE is required to specify the type of declaration. The optional dot (period) indicates that an increment instruction to protect any local variables in the stack should not be generated since this is a low level procedure which does not call any other procedures.

The procedure-identifier is required and may be any allowable BPL identifier. The procedure-identifier will be used within the program (block) to invoke the procedure.

The formal-parameter-list is used to name any formal parameters which may be required by the PROCEDURE. The formal-parameter-list, if used, must be enclosed in parentheses, and multiple parameters must be separated by commas. The maximum number of parameters for any PROCEDURE is 10.

The semicolon following the formal-parameter-list is required.

The VALUE option allows for the declaration of VALUE parameters (as opposed to name parameters). VALUE parameters are actually contained in the stack when a PROCEDURE is entered; whereas name parameters have their addresses in the stack, and are accessed indirectly. VALUE parameters have no meaning outside of the PROCEDURE in which they are declared. The word VALUE, followed by one or more of the named parameters in the formal-parameter-list specifies that those parameters are VALUE parameters. If the value-parameter-list contains more than one entry, they must be separated by commas. A semicolon delimiter is required to terminate the VALUE parameter list.

Parameter-specifications are required if parameters are involved. Each parameter named in the formal-parameter-list must be declared, regardless of whether or not it is included in the value-parameter-list.

The word BEGIN is required to indicate the beginning of the PROCEDURE body.

The procedure-body-declarations are those variables local to the PROCEDURE. Labels used within the PROCEDURE should be declared here. Those declarations not preceded by OWN are stack-relative; that is they are placed in the stack. OWN declarations are segment-relative. Only OWN variables can be pre-initialized. A stack variable cannot be forced to a MOD n address.

The PROCEDURE body is a statement that is to be executed when PROCEDURE is called. This statement may be any of those listed in the syntax of statements and therefore may be a PROCEDURE statement calling upon itself. Procedures may thus be called recursively.

A PROCEDURE body itself must not be labeled. A GO TO statement appearing in a PROCEDURE should not lead outside that PROCEDURE. Branching outside the PROCEDURE without an EXT causes the stack to retain its PROCEDURE state when the program is no longer there. If any statement in a PROCEDURE body is labeled, the declaration of that label must appear in the appropriate block heading within the PROCEDURE body.

The word END is required to terminate the PROCEDURE declaration.

Examples:

```
EXAMPLE:
BEGIN
PROCEDURE A;                                (See note 1.)
BEGIN
    IX1 := 0;
    IX2 := 1;
END;
PROCEDURE B;                                (See note 2.)
BEGIN
INTEGER TEMP (2);
    IX1  := 0;
    TEMP := IX2.UN.+6.2;
END;
PROCEDURE. C (A1, B1, C1)                   (See note 3.)
    VALUE A1, C1;
INTEGER A1 (6);
ALPHA B1(11);
SIGNED INTEGER C1(7);
BEGIN
    OWN INTEGER D1(1):=0;
    IX1 := C1+A1;
    B1 := "I LOVE LUCY";
    D1 := 1
END;
FORWARD PROCEDURE D;                        & (See note 4.)
FORWARD PROCEDURE E(J,K,L,);                & (See note 5.)
    VALUE J, K, L;
    BIT J, K, L;
SEGMENTED PROCEDURE F(X,Y);                 & (See note 6.)
    ALPHA X(6);
    NUMERIC Y(4);
BEGIN
    LABEL L;
    L: X := Y;
END;
PROCEDURE E (J,K,L);                        & (See note 7.)
    VALUE J,K,L;
    BIT J,K,L;
BEGIN
    J := TRUE
END;
PROCEDURE D;                                & (See note 8.)
BEGIN
    DISPLAY "PROC D";
END;
END;
```

Notes:

1. A is a procedure with no formal parameters and no local variables.
2. B is a procedure with no formal parameters but a 2 digit local variable (TEMP, which is stack relative).
3. C is a procedure that is not to have an INC as first instruction to protect local variables. There are 3 formal parameters (2 of which are VALUE parameters) and a segment relative local variables (D1).
4. Simple FORWARD PROCEDURE.
5. E is a FORWARD PROCEDURE declaration of a PROCEDURE having 3 VALUE parameters J, K, L.
6. F is a SEGMENTED PROCEDURE that has 2 name parameters.
7. E is actual declaration of PROCEDURE mentioned in note 5.
8. D is actual declaration of PROCEDURE mentioned in note 4.

SUBROUTINE

The function of a SUBROUTINE declaration is to specify that another Type I ICM will be called from within this Type I ICM, and designates the structure of its parameters.

The format of the SUBROUTINE declaration is:

```
SUBROUTINE subroutine-identifier [ ( formal-parameter-list ) ] ;

    [parameter-specifications ; ]
```

Figure 4-17. Format of SUBROUTINE Declaration

SUBROUTINE is treated by the compiler much as a FORWARD PROCEDURE declaration, in that it merely states that a Type I ICM will be called, and describes its parameters. It does not occupy any memory locations.

The subroutine-identifier is required and may be any allowable BPL identifier. The subroutine-identifier will be used within the Type I ICM to invoke the SUBROUTINE.

The formal-parameter-list is used to name any formal parameters which may be required by the SUBROUTINE. The formal-parameter-list, if used, must be enclosed in parentheses, and multiple parameters must be separated by commas. The maximum number of parameters is 10.

The semi-colon following the formal-parameter-list is required.

Parameter-specifications are required if parameters are involved. Each parameter named in the formal-parameter-list must be declared. All parameters involved with a SUBROUTINE must be name parameters.

The SUBROUTINE declaration is valid only when being compiled to a Type I ICM file and is not permitted otherwise.

Example:

```
BEGIN
@ICM "ANYTNG"                          & Type I ICM DECLARATION
PROCEDURE HERE;
BEGIN
SUBROUTINE THERE (ONE, TWO);           & SUBROUTINE DECLARATION
SIGNED INTEGER ONE (7), TWO (7);       & PARAMETERS
SIGNED INTEGER ABC (7), DEF (7);       & LOCAL VARIABLES
ABC :=IX1;
DEF :=IX2;
THERE (ABC,DEF);                       & SUBROUTINE CALL
END;                                   & OF ICM
@ICM
END;
```

UNSEGMENTED

The UNSEGMENTED declaration is used to inhibit segmentation and its resultant overlay mechanism when encountering a BEGIN followed by declarations.

The format of the UNSEGMENTED declaration is:

```
BEGIN UNSEGMENTED
```

Figure 4-18. Format of UNSEGMENTED

The BEGIN must be matched by a corresponding END to complete the UNSEGMENTED block.

Care must be taken to ensure that the block is not "fallen into" as invalid instructions may result. It is the programmer's responsibility to manage this area.

SECTION 5
EXECUTABLE STATEMENTS/CONTROL AND ASSIGNMENT

GENERAL

Executable statements perform the data transformations and the decision-making functions of a BPL program. For ease of reference, they are described in alphabetical sequence on the following pages.

ACCEPT

The function of the ACCEPT statement is to permit entry of low-volume data through the operator's console (ODT).

The format of the ACCEPT statement is:

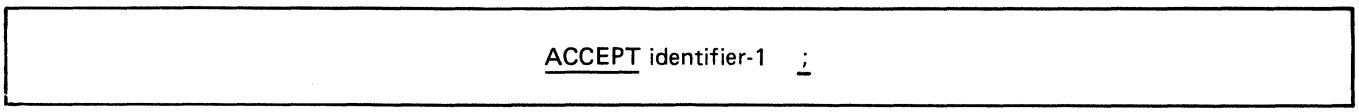


Figure 5-1. Format of ACCEPT Statement

This statement causes the operating object program to halt and wait for appropriate data to be entered on the operator's console (ODT). The ODT entry will replace the contents of memory specified by the identifier. The systems operator answers an ACCEPT halt by keying in the following message:

mix-index AXdata-required

If a blank appears between the AX and data-required, the blank character will be included in the data-stream.

If the number of characters entered exceeds the size of the receiving identifier, the data will be truncated from the right. If the number of characters entered is less than the size of the receiving identifier, an ETX (@03@) will be placed in memory following the last character entered. The number of characters entered may not exceed 60.

Because of the inefficiency of entering data through the keyboard, this technique of data transmission should be solely restricted to low-volume input data.

An indirect field length override on identifier-1 will be ignored.

ACCUMULATOR CONSTRUCTS

A twenty-digit accumulator is provided for fixed-length arithmetic operations. Accumulator load, store, and arithmetic commands consist of a two digit operation code and one address syllable. The Accumulator Manipulate instruction is a four-digit instruction consisting of a two-digit operation code and two variant digits. The functions of the Accumulator Manipulate instruction are:

- Normalize Accumulator
- Convert real to integer
- Set sign of Mantissa to +.
- Set sign of Mantissa to -.
- Complement Sign of Mantissa
- Clear Accumulator to -99+0
- Increment Algebraically the Exponent by the value in the second variant digit.
- Decrement Algebraically the Exponent by the value in the second variant digit.

Accumulator instructions are generated only when CONTROL OP B4700 is specified.

All data referenced by accumulator commands is assumed to be word-aligned, fixed-length data in the form (FIXED INTEGER or FIXED REAL) requested by the instruction, and REAL numbers may be single or double precision, as specified in the address controller of the instruction.

The accumulator commands are associated with an error trap for overflow, underflow, or divide by zero conditions. If the error trap has been enabled by the programmer, error branching and passing required parameters are a by-product of the accumulator instructions.

Accumulator instructions are generated by the compiler as they are needed, as explained in succeeding text. Except as shown, there is no BPL syntax to generate explicit Accumulator Manipulate instructions.

The BPL language provides the following means to use the accumulator:

IACCUM is used to specify usage FIXED INTEGER.
RACCUM is used to specify usage FIXED REAL.
DACCUM is used to specify usage FIXED DOUBLE.

Following are examples of accumulator usages:

```
FIXED INTEGER X,A;
FIXED REAL R;
FIXED DOUBLE DOUBON;
IACCUM := X ;                    & INTEGER LOAD
X := IACCM ;                     & INTEGER STORE
RACCUM := R ;                    & REAL LOAD
IACCUM := RACCUM ;               & CHANGE REAL TO INTEGER
IACCUM := IACCUM + A ;           & INTEGER ADD
RACCUM := RACCUM + 100.0 ;       & REAL ADD +03+10000000
RACCUM := RACCUM * 100 ;         & ACCUMULATOR MANIPULATE
                                 & ADJUST EXPONENT BY + 2
RACCUM := RACCUM / 10 ;          & ACCUMULATOR MANIPULATE
                                 & ADJUST EXPONENT BY - 1
DACCUM := DOUBON ;               & LOAD DOUBLE TO ACCUMULATOR
RACCUM := IACCUM * 1.5 ;         & CHANGE INTEGER TO REAL AND
                                 & REAL MULTIPLY
```

An accumulator name on the right side of an assignment statement states previous usage of the accumulator. No check is made by the compiler to ensure this.

Example:

```
IACCUM := 5 ;                    & THE ACCUMULATOR'S
RACCUM := RACCUM.+3.5;           & USAGE IS INTEGER FROM
                                 & FIRST INSTRUCTION YET
                                 & SECOND INSTRUCTION
                                 & SAYS PRIOR USAGE IS
                                 & REAL
RACCUM := IACCUM.+3.5;           & CORRECT USAGE
```

Multiple instructions may be generated by the compiler to get the individual operands in the same mode etc., for arithmetic operations.

ARM

The function of ARM is to request that the processor error/program exception soft interrupt be enabled. The format of the ARM statement is:

```
ARM ;
```

Figure 5-2. Format of ARM Statement

The ARM statement is most valuable for programs which must provide a graceful termination or take special actions at a processor or program error or at breakout/restart time. If ARM is enabled, a processor error or program exception, defined as follows, will cause the MCP to transfer control to the program's soft interrupt routine. If ARM is not enabled when an error occurs, the program is terminated.

When an exception occurs, the MCP places the following data into the ARMed program:

Location	Contents
BASE.+64.6	Base-relative program address at the time of the interrupt.(Absolute address with MCPs prior to ASR 6.1 MCPVI.)
BASE.+70.3	Base register value
BASE.+73.3	Limit register value
BASE.+76.1	ASCII, overflow, and comparison flip-flops stored as: 8-bit = ASCII 4-bit = Overflow 2-bit = COM L 1-bit = COM H
BASE.+80.4	Result descriptor (see following text)

The program is then reinstated at the address specified in BASE.+94 if that is a valid address. If it is not valid, the program is not terminated.

NOTE

Since the segment dictionary begins at address 64 by default, it must be moved to at least address 100 using the CONTROL DICTIONARY clause.

Three classes of Result Descriptors may be returned to the program if it is ARMed. The first class consists of Pseudo-Processor Result Descriptors indicating hardware-detected errors. The second class consists of Breakout/Restart Result Descriptors indicating a breakout or restart has occurred or that the operator attempted a breakout (BR or BD request) when operator-initiated breakout is inhibited. The third class consists of Preterm Error Codes denoting various software detected program errors.

Processor Result Descriptors are of the form Cn00, where n is a 1-digit integer.

Breakout/Restart Result Descriptors are of the form C0n0, where n is a 1-digit integer.

Pretermination Result Descriptors are of the form 9nn0, where nn is a 2-digit integer.

Result Descriptor numbers are documented in the System Software Interfaces Reference Manual.

The ARM statement complements the current setting of the soft interrupt toggle. If the toggle is off, ARM will turn it on. If it is already on, executing the ARM statement will turn it off. It is the programmer's responsibility to know the ARM status.

A program is no longer ARMed when the ARM branch has been taken.

For additional information concerning the ARM statement, refer to the DISARM statement.

ASSIGNMENT

The assignment statement is used to assign the value of an expression to a specified identifier.

The formats of the assignment statement are:

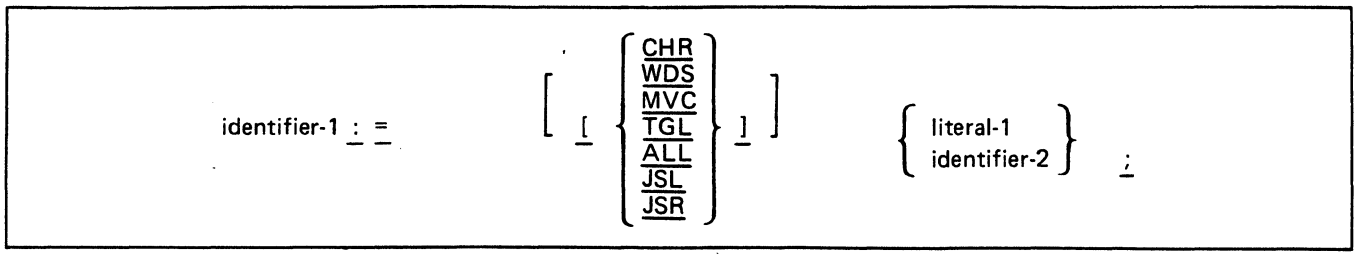


Figure 5-3. Format of the Assignment Statement, Option 1 (MOVE)

The double special character colon-equal (:=) is called the assignment symbol and is read as "is replaced by".

The value of literal-1 or identifier-2 to the right of the assignment symbol is assigned (moved) to identifier-1. Any identifier must have been previously declared before it can be used.

Assignment overrides are used primarily in the simple assignment statement to override (or force) a particular compiler action. The following assignment overrides are available:

Table 5-1. Assignment Overrides

Override	Function
ALL	The [ALL] override indicates that some form of "spreading" is to take place. If indirect addressing is specified for the receiving field, and the size of the receiving field is not an integral multiple of the size of the sending field, a syntax error will result. The entire contents of the sending field (identifier-2) or the literal value (literal-1) will be repeated throughout the entire receiving field. If indirect field length is specified in either the sending or receiving field (or both), the lengthof the receiving field is taken as the number of repetitions of the entire sending field desired; that is, the effective length of the receiving field will be the product of the lengths of the sending and receiving fields. The sending field (or literal) may not be signed, nor may it exceed 100 (bytes or digits) in length.
CHR	The "character" override forces the generation of a "move alpha" (MVA) command. Both addresses must be MOD 2 or a syntax error will occur. A literal (literal-1) is not allowed with this override.
JSL	The "justified left" override has meaning only when it is desired to have a numeric field justified left with zero fill to the right.
JSR	The "justified right" override has meaning only when it is desired to have an alpha-numeric field justified right with blank fill to the left.
MVC	The "move and clear" override is the same as the [WDS] override except that the sending field is set to 4-bit zeros.
TGL	The "toggle" override indicates that the comparison indicator setting following this instruction is significant and the compiler should not attempt to optimize this particular move instruction. An error message will result if a single move instruction cannot complete the operation.
WDS	The "words" override forces the generation of a "move words" (MVW) command. Both addresses must be MOD 4 or a syntax error will occur. If sizes are not equal, a warning is issued and the smaller size is used. The size (or number of words) to be moved is placed right-justified in the digit AF and BF fields of the generated code. If indirect field length is indicated for either operand, these AF and BF fields will be changed. Specifying indirect field length for identifier-2, the sending field, will place the indirect field length in the AF field, which is the number of thousands and hundreds of words to be moved. Specifying indirect field length for identifier-1, the receiving field, will place this indirect field length in the BF field, which is the number of tens and units of words to be moved.

Consider the following example:

```
BEGIN LABEL DOIT;
        INTEGER INFLA (2) = 36;
        INTEGER INFLB (2) = 38;
        INTEGER WORDA (8) MOD 4;
        INTEGER JUNKA (400);
        INTEGER WORDB (8);
        INTEGER JUNKB (400);
DOIT:
        INFLA := 1;
        INFLB := 2;
        WORDA.INFLB := [WDS] WORDB.INFLA;
END:
```

This example will generate a move of 0102 words (408 digits) starting at WORDB and moving to WORDA. A literal (literal-1) is not allowed with this override.

If the sending field size is greater than the receiving field length, OVERFLOWA may occur. A numeric move will not take place and no warning is generated. If the sending field is 100 or more longer than the receiving field, numeric move OVERFLOW results are defined.

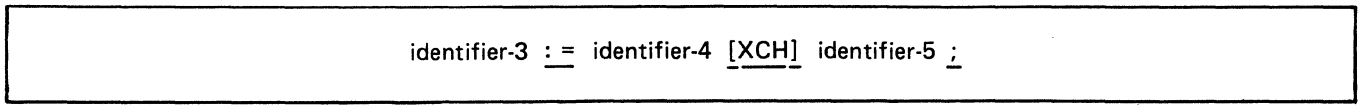


Figure 5-4. Format of the Assignment Statement, Option 2 (EXCHANGE)

The "exchange" option causes the generation of a "move links" (MVL) instruction to "exchange" the contents of the sending and receiving fields (identifier-3 and identifier-4). Both fields must be the same size and type, and must not overlap. The presence of a third identifier (identifier-5) causes generation of a 3-way "move-links" (MVL) instruction to replace the contents of identifier-3 with the contents of identifier-4, replace the contents of identifier-5 with the contents of identifier-3. The [XCH] override may appear anywhere after the receiving field operand and prior to the semicolon, however the above format is recommended. All fields (identifiers) must be the same size and type, and must not overlap.

A move links may not be embedded within an expression.

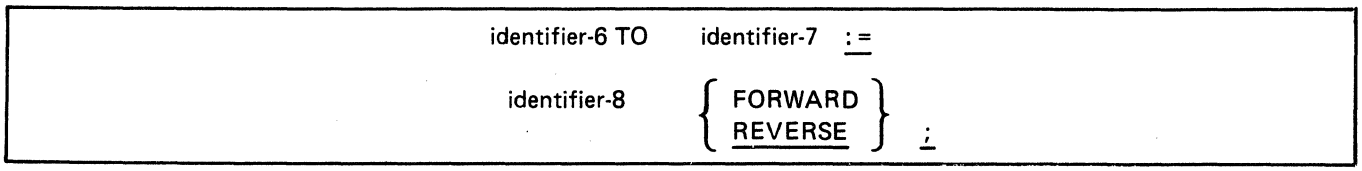


Figure 5-5. Format of the Assignment Statement, Option 3 (MOVE DATA, CONTROL OP B4700 only)

When two identifiers precede the assignment symbol (:=) a "move data" (MVD) instrcution is generated. This instruction is valid only when CONTROL OP B4700 is specified.

A "move data" FORWARD (default) statement will cause data to be moved from identifier-8 into the location beginning at identifier-6 up to the beginning of identifier-7.

A "move data" REVERSE will cause data to be moved from the memory preceding identifier-8 into the area preceding identifier-6 until the lower limit identifier-7 is reached.

All addresses must be MOD 4 or a syntax error will result.

A move data may not be embedded within an expression.

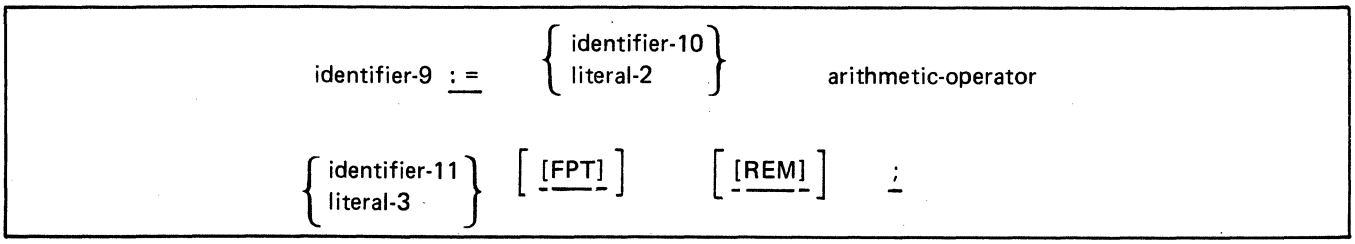


Figure 5-6. Format of the Assignment Statement, Option 4 (COMPUTE)

For more information on option 4, refer to note 1 at the end of the "Assignment" portion of this section.

The preceding format causes the contents of identifier-9 to be replaced by the result of the arithmetic operation performed.

Allowable arithmetic operators are:

Operator	Function
+ (plus sign)	Addition
- (minus sign)	Subtraction
* (asterisk) or MUL	Multiplication
/ (slash) or DIV	Division

In all operations, a literal may be used on either side of the arithmetic operator.

A Boolean operand must not be used in an arithmetic operation

If one operand of an arithmetic operation is FIXED, the other operand must fulfill one of the following requirements:

1. It must be FIXED.
2. It must have an "IA" controller and be pointing to a valid fixed field and defined in (1) or (3).
3. It must be an integer with a mod-4 address and is not a name parameter and is of one of the following sizes:

   a. 7-SN
   b. 8-UN
   c. 11-SN
   d. 12-UN
   e. 19-SN
   f. 20-UN

If the size declared for the result field does not conform to rules for the hardware operation, leading zeroes will be provided, or leading digits dropped, in the code generated by the compiler. This can result in extra instructions and work-areas being generated.

If the user intends to check for overflow after an arithmetic operation, it is his responsibility to assure that overflow is off prior to that operation. If overflow is detected during an operation, the result field is not changed.

When using the DIV operator the length of the dividend must be greater than the length of the divisior.

REMAINDER is a reserved word provided to gain access to the remainder of a divide operation. The length of any remainder is the length of the divisor plus the length of the quotient. Its type is signed numeric. The remainder location is volatile and if needed should be used promptly following a divide operation since any subsequent divides in the same segment will destroy previous results.

If an indirect field length is specified on the divisor and/or the quotient, REMAINDER is not used. Instead, the dividend field is used. A warning is issued by the compiler if this condition exists.

If an indirect field length is specified on only the dividend, the REMAINDER length is the sum of the lengths of the divisor and quotient; however, this may cause a run time overflow condition.

The following assignment overrides are available for use in an arithmetic operation:

Table 5-2. Assignment Overrides in Arithmetic Operations

Override	Function
FPT	The "floating point" override causes generation of a floating point instruction to perform the indicated operation. In a compound expression it applies to the entire expression and all its operations.
REM	The "remainder" override causes the divident of a divide operation (identifier-10) to be replaced with the remainder. The location accessed by the reserved word REMAINDER is not affected if the [REM] override is used. In a compound expression it applies to the entire expression.

NOTE
Overlapping fields in arithmetics generate extra moves to overcome hardware limitations. To prevent extra moves, the REM override may be used.

Compound arithmetic expressions are permitted. Refer to Note 1 at the end of the "Assignment" portion of this section.

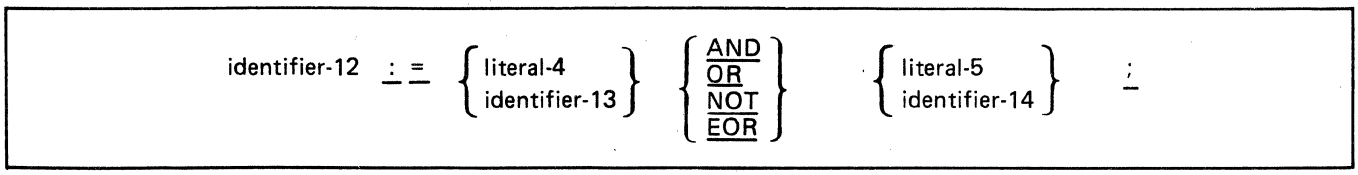


Figure 5-7. Format of the Assignment Statement, Option 5 (LOGICAL OPERATORS or BOOLEAN OPERATORS)

For more information on option 5, refer to note 1 at the end of the "Assignment" portion of this section.

The logical operators assignment type statement is used to manipulate or check individual bits, and is used to generate logical AND, NOT and OR instructions. The operands must both be unsigned integers, or both "numeric" or "alpha". An operand must not be signed, "real", or a Boolean value.

If the two operands have different lengths then they are left-justified. The shorter operand is filled with trailing zeroes for AND and OR, or trailing hexadecimal Fs for NOT and EOR. The data type of the result is the same as that of both the operands.

The logical operator AND will compare the identifier-13 field bits with the corresponding identifier-14 field bits and store a 1 bit into the corresponding identifier-12 field bit if the corresponding identifier-13 and identifier-14 field bits are both on.

The logical OR will compare the identifier-13 field bits with the corresponding identifier-14 field bits and store a 1 bit into the corresponding identifier-12 field bit if either or both of the corresponding identifier-13 and identifier-14 field bits are on.

The logical NOT and EOR will compare the identifier-13 field bits with the corresponding identifier-14 field bits and store a 1 bit into the corresponding identifier-12 field bit if the corresponding identifier-13 and identifier-14 field bits are not equal. Either sending field (but not both) may be a literal. If the sending fieds are not the same length, the shorter field will be assumed to have trailing 4-bit zeros. Indirect addressing may be used, however the final data type of all three fields must be the same, and may not be signed numeric (SN).

Examples:

```
A := @37@ AND B.2UN;
A.IX1 := @1@ OR A.IX1   ; & The @ is not required.
```

Compound expressions are permitted. Refer to Note 1 at the end of the "Assignment" portion of this section.

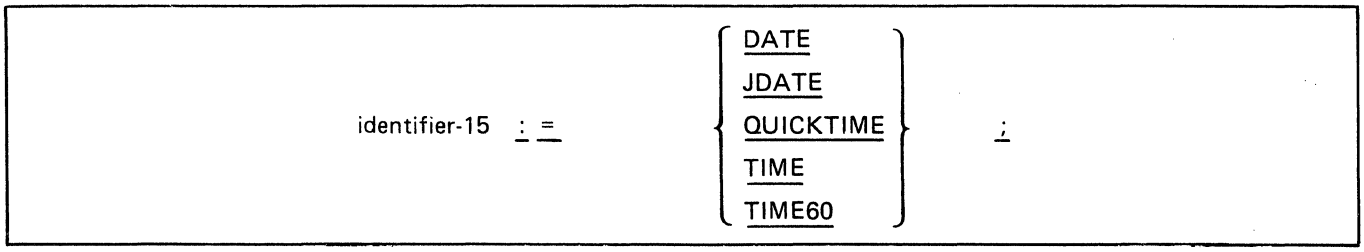


Figure 5-8. Format of the Assignment Statement, Option 6 (SPECIAL BRANCH COMMUNICATES)

For more information on option 6, refer to note 3 at the end of the "Assignment" portion of this section.

The special branch communicate instructions are used to assign values unknown at compile time to identifier-15 at execute time.

The special names involved and their formats are:

Table 5-3. Names of Special Brand Communicate Instructions

Name (Reserved Word)	Format	Function
DATE	6UN MMDDYY	Current Calender Date
JDATE	5UN YYDD	Current Julian Date
QUICKTIME	10UN MMMMMMMMMM	Time of day - milliseconds See note.
TIME	10UN MMMMMMMMMM	Time of day - milliseconds
TIME	6010UN 00HHMMSSss	Time of day Hours, minutes, seconds, 60/seconds

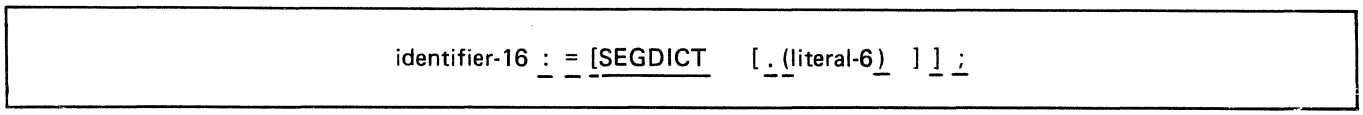


Figure 5-9. Format of the Assignment Statement, Option 7 (SEGDICT)

For more information on option 7, refer to note 3 at the end of the "Assignment" portion of this section.

Identifier-16 will contain the address of the segment dictionary.

The literal-6 option is used to get the address of a specific segment dictionary entry. Default case points to the same Segment Dictionary entry in which the statement appears.

SEGDICT may also be used in an ADDRESS = statement.

NOTE
BPL considers the start of the segment dictionary to be segment 0. The program main block is segment 1.

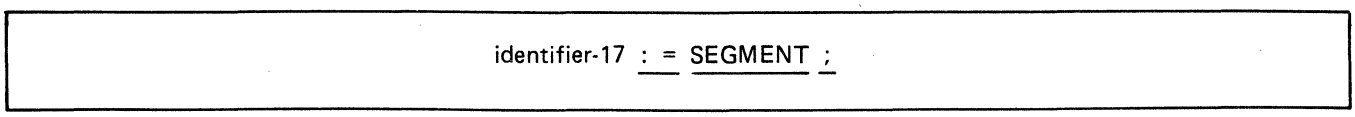


Figure 5-10. Format of the Assignment Statement, Option 8 (SEGMENT)

For more information on option 8, refer to note 3 at the end of the "Assignment" portion of this section.

Identifier-17 will contain the segment number of the current block.

Identifier-17 must be declared INTEGER (4).

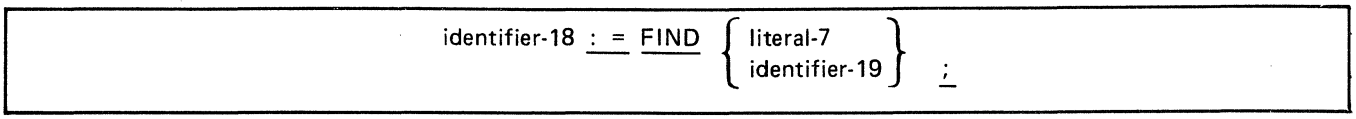


Figure 5-11. Format of the Assignment Statement, Option 9a (INTERROGATE FILE on disk)

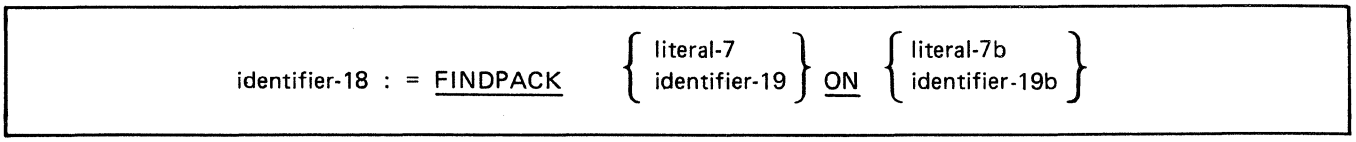


Figure 5-12. Format of the Assignment Option 9b (INTERROGATE FILE on Diskpack)

For more information on options 9a and 9b, refer to note 3 at the end of the "Assignment" portion of this section.

Option 9a requests the MCP to check for the presence of a disk file with a value of ID equal to literal-7 or identifier-19.

Option 9b requests the MCP to check for the presence of a diskpack file with a name equal to literal-7 or identifier-19, on the pack family named by literal-7b or identifier-19b.

Identifier-19 and identifier-19b must be declared ALPHA (6). Literal-7 and literal-7b must be alpha literals of six characters or less.

Identifier-18 must be declared INTEGER (1).

Upon execution, the MCP will place a 0 or 1 in identifier-18. If the file is found in the disk directory, identifier-18 will equal 1; otherwise, it will equal 0.

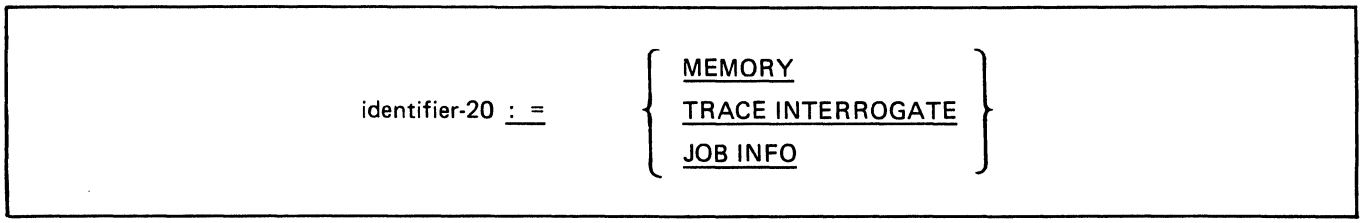


Figure 5-13. Format of the Assignment Statement, Option 10a (PROGRAM PARAMETER BRANCH COMMUNICATES, ANY MCP)

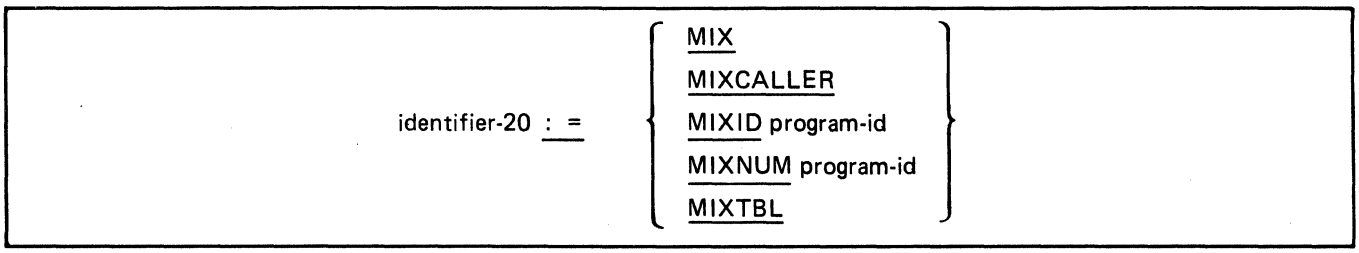


Figure 5-14. Format of the Assignment Statement, Option 10b (PROGRAM PARAMETER BRANCH COMMUNICATES, PRE-MCP/VS 2.0)

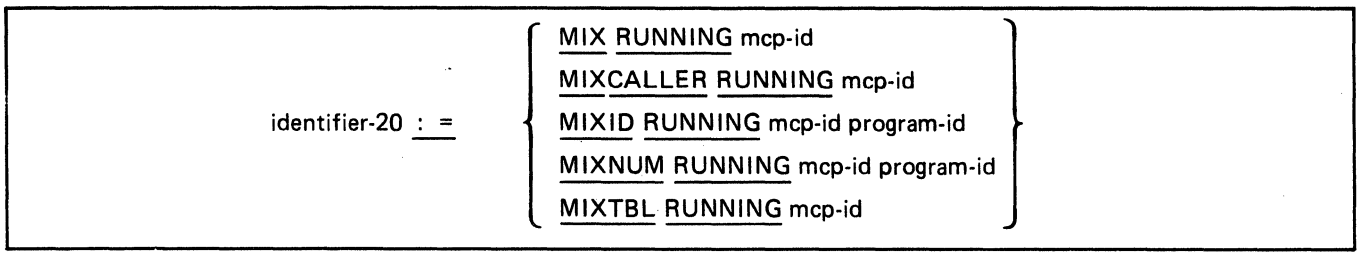


Figure 5-15. Format of the Assignment Statement, Option 10c (PROGRAM PARAMETER BRANCH COMMUNICATES, MCP/VS 2.0 AND LATER)

The program parameter branch communicates are used to determine, at runtime, values concerning the program and its environment.

Table 5-4 lists communicates that can be used with any current MCP.

The special names involved and their formats are:

Table 5-4. Special Names for Use with Any Current MCP

Name	Format	Function
MEMORY	6 UN MMMOOO	Memory assigned to program in thousands of digits. See Note 3.
TRACE INTERROGATE	1 UN	Returns a 1 if the calling program is being traced.
JOBINFO	See Note 4	Allows the programmer to determine information on the environment of the operating system.

Table 5-5 lists communicates that can be used with MCP/VS 1.0 and the previous operating system. Programs using these communicates will not run on MCP/VS 2.0 systems with job mix limits greater than 99. The special names involved and their formats are:

Table 5-5. Communicates

Name (Reserved Word)	Format	Function
MIX	2 UN	Number of programs in the Mix.
MIXCALLER	2 UN	Mix number of calling program.
MIXID program-id	2 UN	Number of programs in the Mix with an identifier indicated by program-id. Program-id must be a six-character alpha literal or an identifier declared ALPHA (6).
MIXNUM program-id	2 UN	Mix number of programs identified by program-id. Program-id must be a six character alpha literal or an identifier declared ALPHA (6).
MIXTBL	See Note 5	Returns information from the MCP mix table to the calling program.

The functions MIX, MIXCALLER, MIXID, MIXNUM and MIXTBL all generate a MIXTBL BCT (see the V Series Program Interfaces Programming Reference Manual). The format of the MIXTBL BCT will vary depending on the operating system.

The mix functions defined in Option 10c require you to specify the MCP name (mcp-id). The code generated by these functions can be transported between MCP/VS 1.0 and MCP/VS 2.0 operating systems. The same program cannot contain statements using option 10b *and* statements using option 10c. A syntax error occurs if the two formats are used in the same program.

MCP-id must be a 17-character alpha literal or an identifier declared ALPHA (17). A value in MCP-id of "MCP" followed by all spaces indicates the 2.0 or later version of MCP/VS. Any other MCP name (such as "MCPIX" or "MCP/VS") indicates an MCP prior to 2.0 MCP/VS.

The special names involved and their formats for Option 10c are:

Table 5-6. Special Names for Option 10c

Name (Reserved Word)	Format	Function
MIX RUNNING mcp-id	4 UN	Number of programs in the Mix.
MIXCALLER RUNNING mcp-id	4 UN	Mix number of calling program.
MIXID program-id RUNNING mcp-id	4 UN	Number of programs in the Mix with an identifier indicated by program-id. Program-id must be a six character alpha literal or an identifier declared ALPHA (6).
MIXNUM program-id RUNNING mcp-id	4 UN	Mix number of programs identified by program-id. Program-id must be a six character alpha literal or an identifier declared ALPHA (6).
MIXTBL RUNNING mcp-id	See Note 6.	Returns information from the MCP mix table to the calling program.

Note 1 (Arithmetic and Conditional Expressions):

Compound arithmetic and conditional expressions are allowed. The operators and their priorities are:

Highest:	+,– (Monadic - acts on only one operand)
	*, /
	+,– (Dyadic - acts on two operands)
	LSS, EQL, LEQ, GTR, GEQ, NEQ
	AND
	OR
	EOR, NOT (Dyadic)
Lowest:	NOT (Monadic)

Examples:

```
A := B + C + D
is equated to
  temp := B + C
  A :=temp+ D
  A := B + C * D
is equated to
  temp := C * D
  A := B+temp
```

If an expression contains operators of the same priority then these operators are evaluated from left to right.

The replacement operator (:=) is also a valid operator. It has higher priority than all operators to its left, and lower priority than all operators to its right.

Example:

```
A := B := C        IS EQUIVALENT TO      B := C;
                                         A := C
```

Parentheses may be used to change these priorities.

Note 2 (All Branch Communicate instructions):

Any identifier in a BCT cannot be indexed.

Note 3 (Memory):

For compilers, the format is 7 UN, MMMXEES. MMM is the memory assigned to the compiler in thousands of digits. X is not used. EE is the disk Eu or ID number specified in the COMPILE command for the code file, S is the SYNTAX flag (S = 1 if COMPILE.....SYNTAX). If identifier-20 is 7 UN, the BPL compiler generates a warning.

Note 4 (JOBINFO):

It is the programmer's responsibility to allocate a response area of sufficient size according to the following format:

2	UA - 00 indicates normal response 04 indicates response area too small
17	UA - MCP name
4	UA - MCP release number
6	UA - MCP patch level
6	UN - MCP version date
2	UN - Processor number of caller
17	UA - Hostname
4	UN - MIX number of caller
1	UN - Batch/TSM Flag (7 or A indicates TSM: all other values indicate Batch)
95	UN - <Reserved>

Note 5 (MIXTBL):

It is the programmer's responsibility to allocate a table of sufficient size according to the following format:

Header

Jobs in mix	3 UN
Memory available	3 UN (mod 1000; first available area)

Body (one entry for each program)

MIX-ID	6 UA (program name)
MIX-MF	6 UA (multi-program name)
MIX-NO	2 UN (mix number)
MIX-BC	1 UN (reserved, always zero)
MIX-CA	3 UN (memory used by job, including disk file headers)

Note 6 (MIXTBL Running mcp-id):

It is the programmer's responsibility to allocate a table of sufficient size according to the following format:

Header

Jobs in mix	4 UN
Memory available	10 UN (mod 1000)

Body (one entry for each program)

Program Id	6 UA
Multi-Program Id	6 UA
Task Number	4 UN
Memory used by task	7 UN (code and data)
Processor priority	1 UN
Memory priority	1 UN
Special program code	1 UN

1 = Program is a generator
2 = Program is DMPALL
5 = Dskout or Pack Squash
6 = Program has DCP or MCS status
7 = Timesharing process
8 = Timesharing Handler
A = Generator in shared area
B = DMS Control Program
C = WFL Handler
D = BNA Handler
E = Program is copy

Program Status Code 2 UN

00 = EXECUTING
01 = COMPILING
02 = WAITING I/O
03 = WAITING CORE-TO-CORE
04 = STOPPED
05 = NO COMPLEX WAIT TABLE SPACE
06 = WAITING COMPLEX WAIT
07 = WAITING STOQUE ENTRY
08 = WAITING STOQUE MEMORY
09 = WAITING STOQUE NAME SLOT
10 = WAITING STOQUE PROCESSING
11 = WAITING TRACE
12 = WAITING OPERATOR ACTION
13 = WAITING MEMORY
14 = SLEEPING

BREAKOUT

The function of the BREAKOUT statement is to specify that a programmatic breakout is to be taken at this point, for possible restart.

The format of the BREAKOUT statement is:

```
BREAKOUT ;
```

See CONTROL BREAKOUT for further explanation. Refer also to the OCS commands BD, BR, and RB in the System Software Operation Guide for MCP/VS 1.0, MCPIX, or MCPVI.

The program must have no DISKPACK files open when a BREAKOUT is executed.

NOTE
BREAKOUT is not permitted under MCP/VS 2.0.

CASE

The CASE statement selectively executes one statement within a case-group of statements.

Two formats for CASE statements are shown in the figures 5-16 and 5-17.

Format 1

```
CASE  [NO]  identifier-1  OF
      BEGIN
            statement-0   ELSE
            statement-1   ELSE
            statement-2   ELSE
                .
                .
                .
            statement-n
      END ;
```

Figure 5-16. The CASE Statement, Format 1

Format 2

```
CASE_ [NO]     identifier-1   OF
      BEGIN
            statement-0      ELSE
            statement-1      ELSE
            statement-2      ELSE
                :
            statement-n-1    ELSE BEGIN
            statement-n
ESAC  ;
```

Figure 5-17. The CASE Statement, Format 2

The word CASE or CASE_ is required, identifying the CASE statement. The NO option, if present, will cause the compiler to omit the validity checking code on the value contained in identifier-1 at execution time.

Statement-0, statement-1, ... statement-n make up the statements in the CASE-group.

At execution time, the value of identifier-1 is examined and used as a selector to choose from the statements in the CASE-group. The statements in the group are numbered from zero (0) to N-1 for N statements, and a single statement in the group is executed, depending on value contained in identifier-1. A value which is greater than the number of statements in the CASE-group will cause the highest value to be assumed, unless the NO option was used. In this case, a value out of range will not be detected and the results will be unpredictable.

Any valid BPL statement, including nested CASE statements, DO-group statements, and IF-THEN-ELSE statements and blocks are allowed and are counted as single statements.

CASE statements consisting of only "ELSE GO TO..." will generate more efficient code.

The CASE variable is limited to a maximum of 6 digits. Code and data space is slightly optimized for a two to six digit variable. The CASE statement destroys the previous value in IX1, leaving IX1 negative.

CASE_ differs from CASE in that it must be terminated by ESAC. ESAC provides a visible end to the CASE_ statement, and functions as both an implicit END END and an implicit branch label.

Since ESAC is an implicit END END, an END must not be coded for the BEGIN at the start of the CASE_ statement. However, since ESAC is two implicit ENDs, a BEGIN must be coded before the final statement in the case-group (statement-n), even if that is a single statement.

ESAC is an implicit branch label, permitting the verb EXITCASE to be used in a CASE_ statement. EXITCASE causes an immediate branch to the first executable statement following ESAC. EXITCASE must be surrounded by BEGIN and END.

Refer also to IN_, OUT_, and ELSE_ in Appendix G.

Examples:

```
CASE I OF
      BEGIN
            GO TO LO ELSE          & IF I = 0 THEN GO TO LO
            GO TO L1 ELSE          & IF I = 1 THEN GO TO L1
            GO TO L2 ELSE          & IF I = 2 THEN GO TO L2
            GO TO INVALIDL         & IF > 2 ITS INVALID
      END ;                        & END OF CASE STATEMENT

CASE NO ABC OF                     & NO RANGE CHECK ON ABC
      BEGIN
            GO TO LO ELSE          & IF I = 0
            PROC ELSE              & IF I = 1, CALL PROC (A
                                   & PROCEDURE)
            BEGIN                  & IF I = 2 DO
              A : = B ;            & THIS
              C : = D ;            & COMPOUND
            END   ELSE             & STATEMENT

            GO TO INVALIDL         & IF I = 3 ITS INVALID
                                   & IF I > 3 RESULTS
                                   & UNSPECIFIED
      END ;                        & END OF CASE STATEMENT

CASE_ B OF
      BEGIN
            DISPLAY "0"    ELSE
            DISPLAY "1"    ELSE BEGIN
            EXITCASE   END ELSE BEGIN
            DISPLAY "3"
ESAC;
```

CLOSE

The function of the CLOSE statement is to communicate to the MCP that the designated file-name being operated on or created is programmatically completed, and to fulfill the stated action requirements.

The format of the CLOSE statement is:

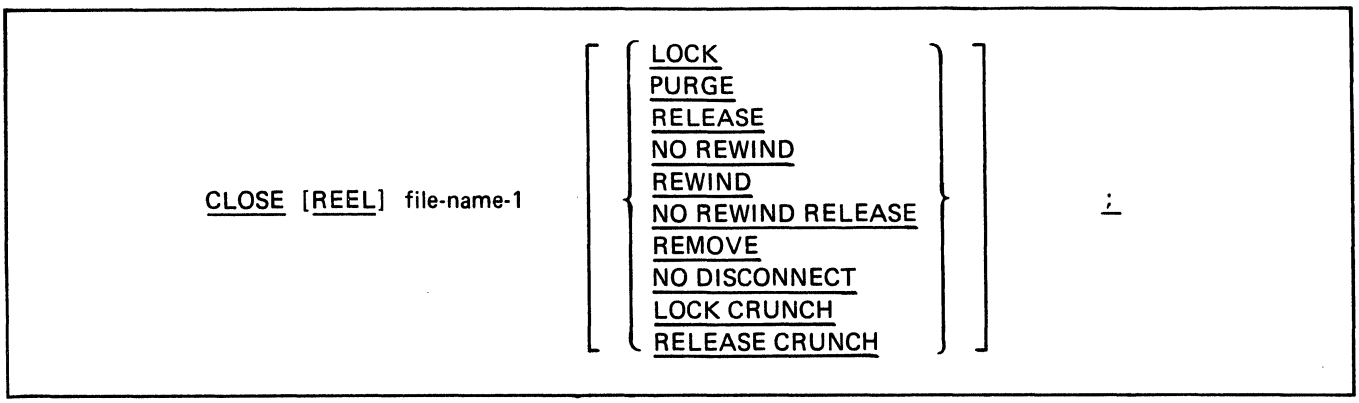


Figure 5-18. Format of the CLOSE Statement

File-names must not be those defined as being SORT files.

A file must have been OPENed previously before a CLOSE statement can be executed for that file.

This statement applies to the following categories of input and output files:

- Files whose input and output media involve print files, card files, etc.
- Files which are contained entirely on one reel of magnetic tape and are the only files on that reel.
- Files which may be contained on more than one physical reel of magnetic tape. Furthermore, the number of reels might possibly be higher than the number of physical tape units provided on the system.
- Disk files.

To show the effects of the CLOSE options, each type of file will be discussed separately.

- Card and MICR Input.
  - CLOSE - releases the input areas, but does not release the reader.
  - CLOSE NO REWIND - same as CLOSE.
  - CLOSE RELEASE - releases the input areas and returns the reader to MCP.
  - CLOSE LOCK - same as CLOSE WITH RELEASE.
  - CLOSE PURGE - same as CLOSE WITH RELEASE.
- Card Output.
  - CLOSE - punches the trailer label (if any), releases the output areas, but does not release the punch.
  - CLOSE NO REWIND - same as CLOSE.
  - CLOSE RELEASE - releases the output areas and returns the punch to the MCP.
  - CLOSE LOCK - same as CLOSE WITH RELEASE.
  - CLOSE PURGE - same as CLOSE WITH RELEASE.
- Tape Input
  - CLOSE - checks the trailer label (if any) and rewinds the tape. It does not release input areas, and the unit remains assigned to the program.
  - CLOSE NO REWIND - same as CLOSE except the tape is not rewound.
  - CLOSE LOCK - releases the input areas, checks the trailer label (if any), rewinds the tape, and the MCP marks the unit not ready.
  - CLOSE RELEASE - releases the input areas, checks the trailer label (if any), rewinds the tape, and returns the unit to the MCP.
  - CLOSE PURGE - releases the input areas, checks the trailer label (if any), rewinds the tape, and if a write ring is in the reel, over-writes the label, making the tape a scratch tape which becomes a candidate for use by the MCP. The unit is returned to the MCP.
  - CLOSE NO REWIND RELEASE - same as CLOSE RELEASE except the tape is not rewound.

- Tape Output.
  - CLOSE - writes the trailer label (if any), and rewinds the tape. The unit remains assigned to the program.
  - CLOSE NO REWIND - writes the trailer label (if any). The tape remains positioned beyond the trailer label (or tape mark if there is no trailer label). The unit remains assigned to the program.
  - CLOSE LOCK - releases the output areas, writes the trailer label (if any), rewinds the tape, and the MCP marks the unit not ready.
  - CLOSE RELEASE - releases the output areas, writes the trailer label (if any), rewinds the tape, and returns the unit to the MCP.
  - CLOSE PURGE - releases the output areas, writes the trailer label (if any), rewinds the tape, returns the unit to the MCP, and the MCP over-writes the label making it a scratch tape, which makes it a candidate for use by the MCP.
  - CLOSE NO REWIND RELEASE - same as CLOSE RELEASE except the tape is not rewound.
- Printer and Lister Output.
  - CLOSE - prints the trailer label (if any), releases the output areas but does not release the printer or lister.
  - CLOSE NO REWIND - same as CLOSE.
  - CLOSE RELEASE - releases the output areas and returns the printer or lister to the MCP.
  - CLOSE LOCK - same as CLOSE WITH RELEASE.
  - CLOSE PURGE - same as CLOSE WITH RELEASE.
- Disk Files. The actions taken on files assigned to DISK will be discussed in terms of old files and new files. An old file is one that already exists on disk and appears in the MCP Disk Directory. A new file is one created by the program and does not appear in the Directory. A new file may only be referenced by the program which creates it.
  - CLOSE

    For an old file, the file is left in the Directory and remains assigned to the program (a subsequent OPEN by the program does not require a disk directory search for the file.

    For a new file, the file is not entered in the Directory, however, it remains on the disk and may be OPENed again by this program.
  - CLOSE NO REWIND - not permitted on disk files.

- CLOSE RELEASE.

  For an old file, the file is left in the Directory, and is available to other programs (a subsequent OPEN by the program requires a Directory search for the file).

  For a new file, the file is entered in the Directory (thereby making it an old file). The file is available to be OPENed by any program.

- CLOSE LOCK.

  For an old file, the file remains in the Directory and is made available.

  A new file is entered in the Directory. Subsequent action is identical to an old file.

- CLOSE PURGE.

  An old file is immediately removed from the disk and deleted from the Directory.

  A new file will be immediately removed from the disk.

- CLOSE REMOVE.

  An old file with the same name on disk is removed from the Directory and the new file is entered in the Directory.

- CLOSE LOCK CRUNCH.

  No effect on old file.

  On a new file, all unused disk will be returned to the MCP.

- CLOSE RELEASE CRUNCH.

  Same as CLOSE LOCK CRUNCH.

• Remote Devices (Data Communications).

- CLOSE - releases the input areas, but does not release the adapter.

- CLOSE RELEASE - releases the input areas and returns the remote device to the system.

- CLOSE NO DISCONNECT - the file is released to the system, but the line is not disconnected.

If a file has been specified as being OPTIONAL, the standard END-OF-FILE processing is not permitted whenever the file is not present.

If a CLOSE statement without the REEL option has been executed for a file, a READ, WRITE, or SEEK statement for that file must not be executed unless an intervening OPEN statement for that file is executed.

The CLOSE REEL option signifies that the file-name being CLOSEd is a multi-reel magnetic tape input/output file. The reel will be CLOSEd at the time of encountering the CLOSE REEL statement and an automatic OPEN of the next sequential reel of the multi-reel file will be performed by the MCP.

COMMENT

The function of COMMENT is to allow the programmer to write explanatory statements in his program which are to be produced on the source program listing for documentational clarity.

The format of the COMMENT statement is:

```
COMMENT [any statement] ;
```

Figure 5-19. Format of the COMMENT Statement

Any combination of the characters from the allowable character set may be included in the character string excluding the semicolon (;).

If an ampersand (&) appears in a source image, the remaining information (through column 72) in that record is COMMENT.

COMMENT may not be used following the reserved word "DEFINE" and before the equal sign (=) in a define.

COMPARE

The function of the COMPARE is to generate a compare or test, with no branch following. Figures 5-20 and 5-21 show the formats for the COMPARE statement.

Option 1

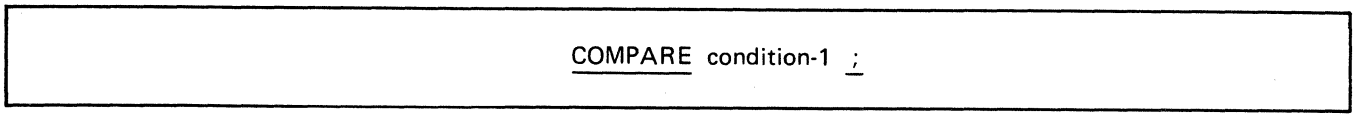


Figure 5-20. Format of the COMPARE, Option 1

This construct is primarily designed for those who require a test to occur, with the appropriate setting of the comparison indicator, but who do not want any kind of conditional branching. The COMPARE statement may be described as an IF statement without any THEN action.

The compiler will not optimize this instruction to a BOT when comparing for zero.

Condition-1 may be any expression containing a relational-operator or boolean-operator as defined for the IF statement option 1 and option 2.

Option 2

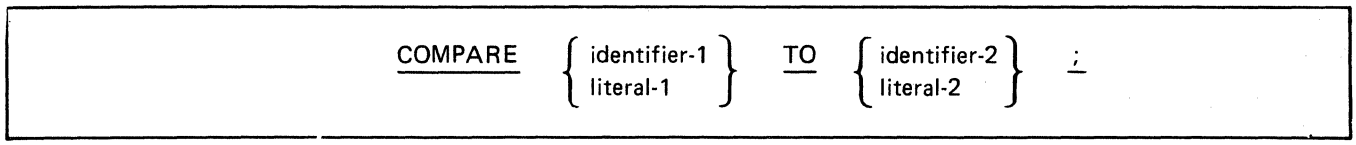


Figure 5-21. Format of the COMPARE, Option 2

This construct allows a COMPARE without stipulating the relational operator. A COMPARE literal-1 TO literal-2 is not allowed.

COPY

The function of COPY is to allow library routines contained on a source language library file to be incorporated into the program.

The format of the COPY statement is:

```
COPY "library-name" ;
```

Figure 5-22. Format of the COPY Statement

A single COPY statement may contain only one library-name. The library-name is bounded by quotes and may not contain more than 6-characters. The library-name is the external (disk directory) name of the library file.

The library file is inserted in the source program immediately after the COPY statement at compilation time. The result is the same as if the library data were actually a part of the source program.

Library data can encompass an entire procedure which may be any number of statements.

Library files may not contain COPY statements.

The COPY construct is completely free form and may be surrounded by BPL statements on the same symbolic record. Any merging of patches will stop until the entire file has been copied.

A library file may be created by inserting the @LIBR compiler directing statement in the source text, or by any program which writes 80-character source records, blocked 5 or 9, to disk. When using @LIBR, the records placed into the library file are simultaneously compiled into the program. See the @LIBR statement under Compiler Directing Statements.

Library files copied from the library are flagged on the output listing by a counter preceded by an "L". The counter will not be initialized by successive copy statements within a single program.

DISARM

The function of the DISARM statement is to request that the processor error soft interrupt for the program be disabled. The format of the DISARM statement is:

```
DISARM ;
```

Figure 5-23. Format of the DISARM Statement

For further information, refer to ARM.

NOTE
ARM and DISARM generate the same code, the effect of which is to toggle the processor error soft interrupt event for the given program.

DISPLAY

The function of DISPLAY is to provide for the printing of low-volume data, error messages, and operator instructions on the console SPO or OCS display. Figures 5-24 and 5-25 show the formats for the DISPLAY statement.

Option 1

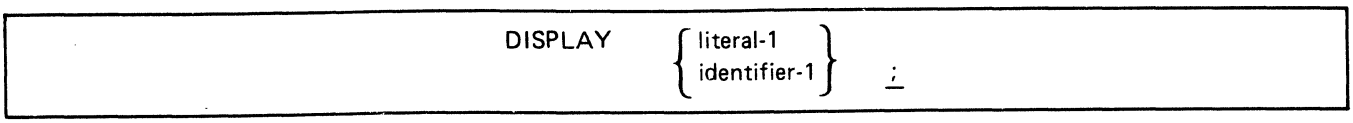


Figure 5-24. Format of DISPLAY, Option 1

The option 1 DISPLAY statement causes the contents of literal-1 or identifier-1 to be written, preceded by the program identification (<P-ID> = <mix-no>).

Identifier-1 may be subscripted and can be declared as an INTEGER or ALPHA item.

A maximum of 60 digits/characters can be DISPLAYED with one statement in option 1 if the program is executed on disk. The limit is 50 if executed on pack, due to the longer program identifier ("on <pack-id>").

Option 2.

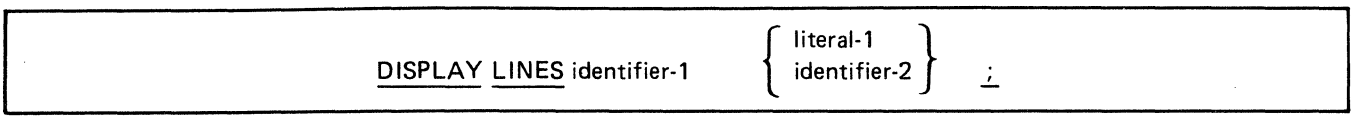


Figure 5-25. Format of DISPLAY, Option 2

The option 2 DISPLAY LINES statement is used for multiline messages to ensure contiguity of all lines.

Identifier-1 may be subscripted, and can be declared as an INTEGER or ALPHA item.

Literal-1 is a numeric literal not exceeding 3 digits. Identifier-2 is an integer data item not exceeding 3 digits. Literal-1 or identifier-2 specifies the number of lines (up to 999) to be displayed.

Each line must be 72 or fewer bytes in length. Lines less than 72 bytes must be delimited by a NULL (&00&) character.

The program identification (<P-ID> = <mix-index>) is not written on the ODT display.

An indirect field length override on identifier-1 or identifier-2 will be ignored.

DO

DO causes a statement or set of statements to be executed repetitevely. Figures 5-26 and 5-27 describe the formats of the DO statement.

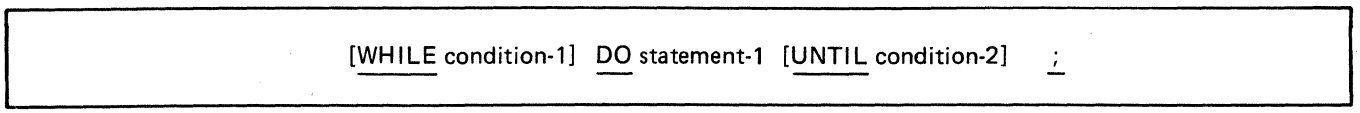


Figure 5-26. Format 1 for DO

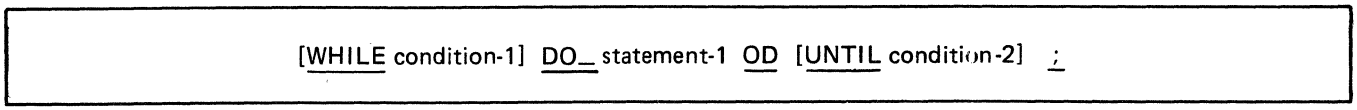


Figure 5-27. Format 2 for DO

Statement-1 represents either a single statement or multiple statements. If there are multiple statements, they must be surrounded by a standard BEGIN...END when using Format-1. When using Format-2, the BEGIN...END areimplicitly provided by the compiler.

Statement-1 is executed repetitevely and indefinitely (in a loop) until some programmatic action forces an exit from the loop, unless a WHILE clause prevents the DO from being executed.

If WHILE is specified, the loop is executed while condition-1 is true. The WHILE is evaluated before the loop is entered. (That is, WHILE condition-1... is equivalent to IF condition-1 THEN... .) Thus, if condition-1 is false when the statement is first executed, the DO loop is never entered.

If UNTIL is specified, the loop is executed until condition-2 becomes true. The UNTIL is evaluated after the loop is executed. Thus, if only an UNTIL clause is present, the loop is executed at least once.

Both WHILE and UNTIL clauses may be specfied. In that case, the loop is entered if condition-1 is true, and is terminated when either condition-1 becomes false or condition-2 becomes true.

Format-2 differs from Format-1 in the following ways:

- A BEGIN...END is implicitly provided around statement-1.
- DO_ includes an implicit branch label, permitting the TOPLOOP verb to be used.
- DO_ must be followed by OD. This provides a visible end to the loop on the program listing.
- OD includes an implicit branch label, permitting the EXITLOOP verb to be used.

The loop can be exited in four ways:

1. If WHILE is specified, the loop ends when condition-1 becomes false.
2. If UNTIL is specified, the loop ends when condition-2 becomes true.
3. The program can branch to a label outside of the DO. Such a label can be specified in a GO or in the action-label portion of a statement permitting such a label (such as READ or FILL).
4. With DO_...OD (not DO), the verb EXITLOOP causes control to be transferred to the first executable statement after the next OD. (If UNTIL is present with that OD, UNTIL is bypassed.)

If statement-1 consists of multiple statements, all of those statements are executed in one execution of the loop mentioned in exit conditions 1 and 2 above.

Condition-1 and condition-2 conform to the rules for conditions under the IF statement. Refer to IF.

When using Format 2 (DO_...OD) the verbs TOPLOOP and EXITLOOP can be used within statement-1.

TOPLOOP causes control to be transferred to the first executable statement in statement-1.

EXITLOOP causes an immediate branch out of the loop.

Examples:

```
DO A := A + 1          or      DO_ A:=A + 1
UNTIL A = 9;                   UNTIL A = 9;
```

The above statement is equivalent to:

```
L1: A := A+1 ;
IF A NEQ 9 GO TO L1;
```

Statement-1 can be a compound statement bounded by BEGIN/END.

For example

```
WHILE Y < 500                    or      WHILE Y < 500
DO    BEGIN                              DO_
          A := X*2;                             A := X * 2;
          Y := Y+A;                             Y := Y + A;
   END                                    OD
UNTIL A = 100;                           UNTIL A = 100;
```

The above state is equivalent to:

```
L1:     IF Y LSS 500 THEN
        BEGIN
            A := X*2;
            Y := Y+A;
            IF A NEQ 100 GO TO L1;
        END;
```

Condition-1 is tested before execution of the DO loop and condition-2 is tested after execution of the DO loop.

Although the UNTIL clause does not have to be explicitly stated, the programmer must provide some mechanism for leaving the DO loop. The UNTIL clause may be contained in statement-1.

For example:

```
DO                                  or          DO_
  BEGIN READ X [EOF];                               READ X [EOF];
        WRITE X;                                    WRITE X;
  END;                                          OD;
EOF:                                        EOF:
```

This is a valid DO statement with a self-contained UNTIL in the READ statement.

This example illustrates TOPLOOP and EXITLOOP. The statements on the right are equivalent to those on the left.

```
WHILE <condition>
DO_
    <statement>
    :
    IF <condition> THEN
        EXITLOOP;
    :
    IF <condition> THEN
        TOPLOOP;
    :
OD
UNTIL <condition>;
```

```
LABEL A, B:
WHILE <condition>
        A:
              DO    BEGIN
                    <statement>
                    :
              IF <condition> THEN
                          GO B;
                    :
              IF <condition> THEN
                    GO A :
              :
              END
        UNTIL <condition> ;
    B :
```

DOZE

The function DOZE will cause the suspension of an executing object program for a specified number of seconds.

The format of the DOZE statement is:

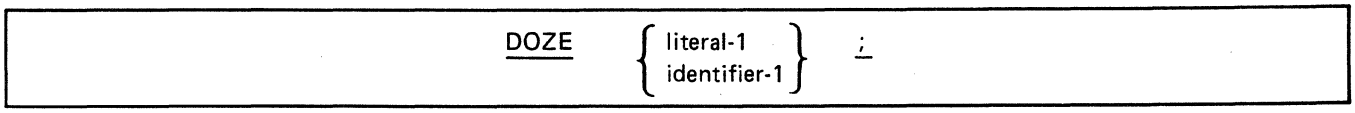


Figure 5-28. Format of the DOZE Statement

A DOZE statement specifying a literal will cause the executing object program to be suspended for that number of seconds and to automatically become reinstated, after the specified period of time has expired, by the MCP.

The DOZE statement is particularly effective in continuous polling loops where polling is required every few seconds, thus releasing the intervening time to the other programs in the mix.

If identifier-1 is specified as containing the DOZE value, it must be an INTEGER field of 5 digits or less.

The maximum DOZEing period is 23 hours, 59 minutes, and 59 seconds (86,399 seconds).

DUMP

The DUMP statement causes the contents of the specified memory locations to be dumped to a line printer or disk.

The format of the DUMP statement is:

```
DUMP [DISK] [identifier-1    [TO identifier-2] ]  ;
```

Figure 5-29. Format of the DUMP Statement

DUMP with no data-name option causes a complete program DUMP.

DISK specifies a DUMP to head-per-track disk.

DUMP identifier-1 will cause at least 1000 digits to be dumped including the complete value of identifier-1. The number of digits dumped will always be MOD 1000.

DUMP identifier-1 TO identifier-2 will give a dump beginning at the MOD 1000 address containing identifier-1, and will occur in 1000 digit segments to include all digits between the identifiers indicated. Identifier-1 and identifier-2 may be program labels, however as no length is associated with a label the dump will begin with 1000 digit area containing the start of identifier-1, and will terminate with the 1000 digit area containing the start of identifier-2.

After the DUMP, execution of subsequent instructions continues normally.

Examples:

```
DUMP ;
DUMP X;
DUMP X TO LAB ;
DUMP DISK ;
```

EDIT

The function of the EDIT statement is to move and edit data, usually for printing on a line printer.

The format of the EDIT statement is:

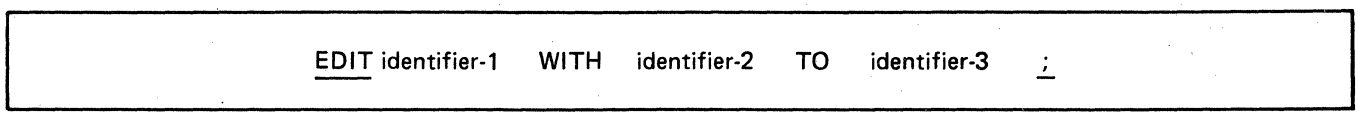


Figure 5-30. Format of the EDIT Statement

The contents of identifier-1 will be edited through the mask contained in identifier-2 and the result will be placed in the field specified by identifier-3. Identifier-2 must reference an ALPHA field, and should usually reference a field created with the PICTURE declaration. Identifier-3 must reference an ALPHA (or 8-bit) field.

Examples:

```
EDIT A WITH PICT TO B ;
EDIT A PICT B ;
EDIT A WITH Q.UA.3 TO B;
```

ENTER

The function of the ENTER statement is to cause control information and parameters to be copied into the subroutine stack, and transfer control to a specified address.

The format of the ENTER statement is:

```
ENTER identifier-1    [WITH literal-1]  ;
```

Figure 5-31. Format of the ENTER Statement

Identifier-1 is the name of the subroutine, procedure or label to be ENTERed.

Literal-1 (if used) is the number of bytes of parameters to be passed. These parameters must be located immediately following the ENTER instruction (using the STORE instructions).

The setting of the comparsion and overflow indicators are stored, then cleared by this instruction.

The parameters may be referenced by using the contents of IX3 plus 16, since the IX3 value points to the beginning of the current stack entry and linkage information occupies the first 16 positions of the entry.

NOTE

It is the programmer's responsibility to provide a corresponding EXIT statement for each ENTER, if entering a label. If the programmer has entered a procedure or subroutine the compiler will generate an EXIT.

EXIT

The function of EXIT is to allow branching from an ENTERed subroutine or procedure, thus reversing the stack action taken upon entry to the subroutine or procedure.

The format of the EXIT statement is:

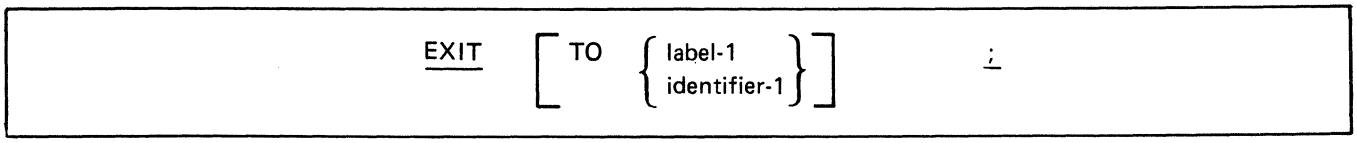


Figure 5-32. Format of the EXIT Statement

An address following the EXIT statement is optional. If omitted, control is transferred to the address specified in the first six digits of the stack entry (the address of the first instruction following the subroutine call). If label-1 is specified, control will be returned to the address of label-1, and the current entry in the stack will be removed.

No validity checking will be performed on the address specified by label-1, or identifier-1, therefore it is the responsibility of the programmer to ensure that control is returned to a valid address.

If a branch or an EXIT is coded immediately before a procedure END, the implied exit is not produced. If an EXIT is coded before an ELSE of an IF statement, the implied branch is not produced.

EXITBLOCK

The EXITBLOCK statement causes an immediate branch out of the range of the current BEGIN_ ...END block. Figure 5-33 shows the format of this statement.

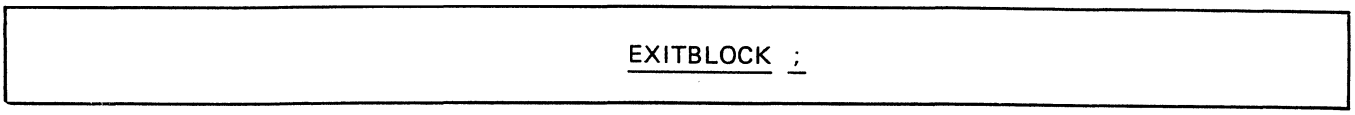


Figure 5-33. Format of the EXITBLOCK Statement

EXITBLOCK is permitted anywhere within a statement block which begins with the reserved word BEGIN_. When EXITBLOCK is executed, it causes control to be transferred to the first executable statement following the END which corresponds to the first BEGIN_ preceding that EXITLOOP.

EXITCASE

The function of the EXITCASE statement is to return control from within a CASE_ statement through a mechanism other than the normal completion of the statement.

Figure 5-34. Format of the EXITCASE Statement

EXITCASE is permitted only with the CASE_ statement. It is not permitted with the CASE statement. EXITCASE can appear anywhere within the CASE_ statement and when it is executed, control is transferred to the first executable instruction after the ESAC statement.

EXITCOND

The function of EXITCOND is to permit an early exit from a conditional statement before the complete processing of the statement is finished.

```
EXITCOND ;
```

Figure 5-35. Format of the EXITCOND Statement

EXITCOND is permitted anywhere within an IF_ statement. It causes control to be passed to the first statement following the FI.

Refer to IF_ (Format 2 of the IF statement) for further details.

EXITLOOP

The EXITLOOP statement causes an immediate branch out of the range of a DO_ statement. Figure 5-36 shows the format of this statement.

```
EXITLOOP ;
```

Figure 5-36. Format of the EXITLOOP Statement

EXITLOOP is permitted only with the DO_ version of the DO statement. EXITLOOP can appear anywhere within the statement loop. When executed, it causes control to be transferred to the first executable statement after the OD statement.

Refer to DO for further details.

EXITROUTINE

EXITROUTINE causes an exit from a file use routine. Figure 5-37 shows the format of this statement.

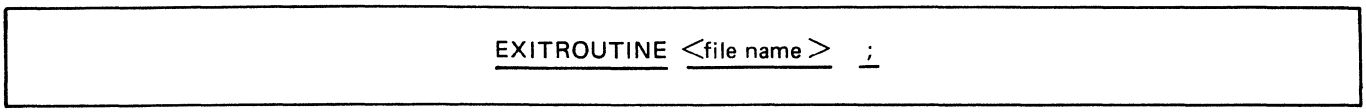


Figure 5-37. Format of the EXITROUTINE Statement

A use routine is any of those specified in a ROUTINE clause in a FILE declaration.

EXITROUTINE must be the last statement executed in a file use routine. When executed, it causes control to be passed to the first statement following the I/O statement which causes the branch to the use routine.

EXITROUTINE causes the program to be reinstated at the address in FIBRCW of the first buffer status block for the file. Consequently, a use routine must not perform an action which can result in entry to another use routine, or the return linkage can be destroyed.

Refer also to ROUTINE under the FILE declaration.

FILL

The function of the FILL statement is to pass data from one program to another when both programs are operating in the same multiprogramming mix.

Figure 5-38 shows the format of the FILL statement.

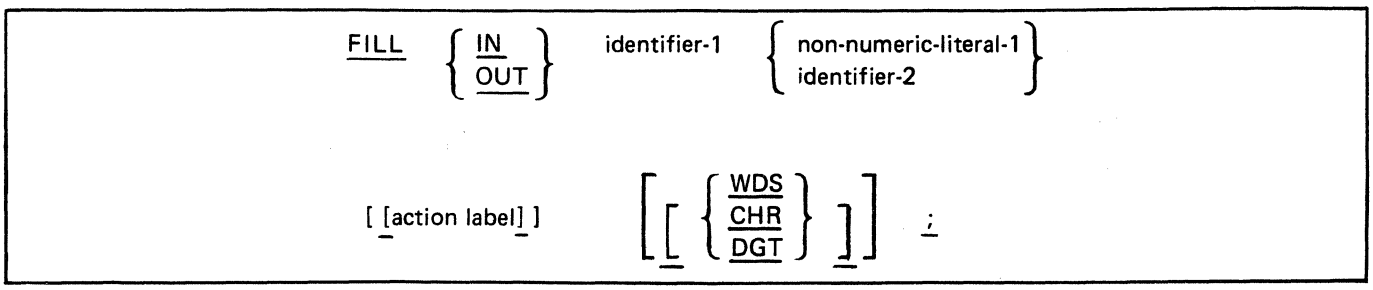


Figure 5-38. Format of the FILL Statement

The MCP Core to Core (CRCR) option must be set "ON" when an object program containing the FILL verb is being operated under the control of a version of the MCP prior to the MCP/VS 2.0.

FILL OUT is the data-sending construct whereby a program using this statement can converse from a self-contained data-name, with another operating program in the same multiprogramming mix. The size of identifier-1 is restricted only by the amount of memory required by the programs themselves. Identifier-2 must be declared as a 6 byte field (or literal) which specifies the program-identifier of the receiving program as reflected in the MCP Program Directory. The receiving program must be in the MCP mix. If the literal-1 is "bbbbbb" (blank), it specifies that any number of receiving programs are to become eligible for the transmission of data.

The action label branch, when specified, will be taken when there is no receiving program ready to receive a transmission. If the action label clause is not used, the program will wait until the FILL has been completed, before proceeding to the next instruction.

FILL IN is the data-receiving construct whereby a program using this statement can receive data from a sending program (identifier-2) into a self contained field (identifier-1). The sending program must be in the MCP mix. If literal-1 is "bbbbbb" (blank), it specifies that any number of sending programs are to become eligible for the transmission of data.

The action label branch, when specified, will be taken if the sending program is not ready to transmit. The data types of the sender and receiver must match.

Reference should be made to the DATACOMM FILL verb located in the Data Communications section of this manual.

GO

The function of GO is to alter the normal flow of the program by transferring control to another location in the program and continuing execution from that point.

Figure 5-39 shows the format of the GO statement.

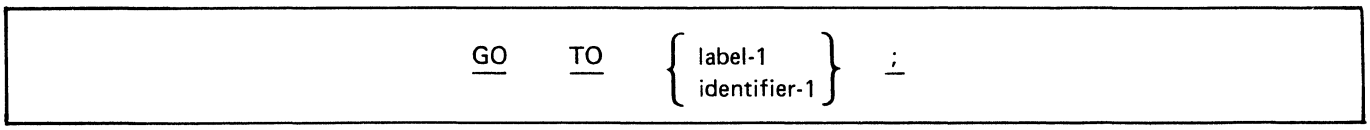


Figure 5-39. Format of the GO Statement

Two restrictions apply to GO statements:

- A GO statement within a procedure cannot refer to a label or identifier outside that procedure.
- A GO statement within a block cannot refer to a label or identifier outside that block.

Except for the above restrictions, no validity checking will be performed on the address specified by label-1 or identifier-1, therefore it is the responsibility of the programmer to ensure that control is passed to a valid address.

IF

The function of the IF statement is to allow a program to select between alternate paths depending on the results of a test.

Figures 5-40 and 5-44 show the two formats you can use for IF statements.

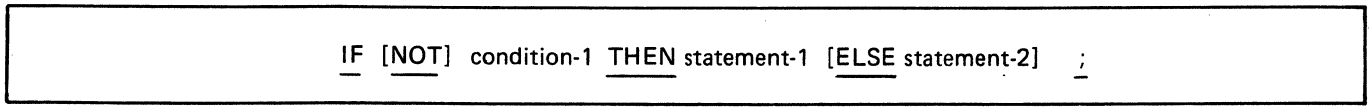


Figure 5-40. Format 1 for the IF Statement

NOT may be placed immediately after the IF thus reversing the comparison results.

Condition-1 may be represented in three ways, described by Options 1, 2, and 3 below.

The word THEN is required; if missing, a syntax error will result.

Statement-1 can be any BPL statement, procedure, or block. This statement will be executed if the condition tested for is met (true).

The optional ELSE condition (statement-2) will be executed if the condition tested for is not met (false).

The options shown in figures 5-41 through 5-43 describe the condition (condition-1) that may be tested with the IF statement. (Refer to Note 4 under ASSIGNMENT for compound expressions.)

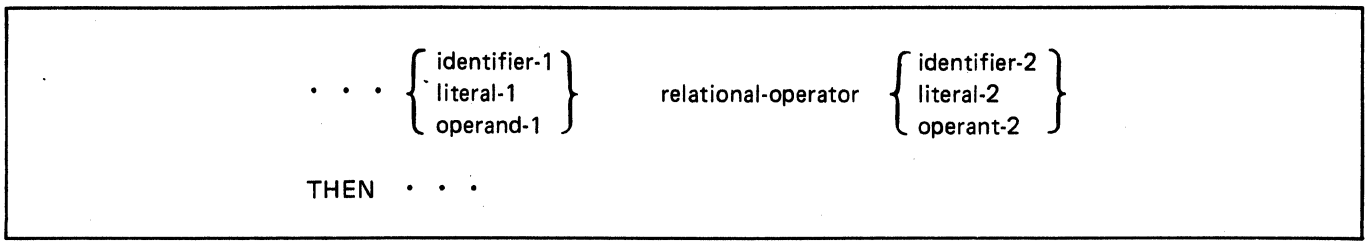


Figure 5-41. Test of Condition-1 with IF, Option 1

The result of an arithmetic operation can be used as an operand (for example, IF A + 1 = B*3 THEN ...).

Either operand may be a literal, but not both.

Relational operators are:

Table 5-7. Relational Operators

Operator	Function
= (or EQL)	Test for equal
–= (or NEQ)	Test for unequal
< (or LSS)	Test for less than
<= (or LEQ)	Test for less than or equal
> (or GTR)	Test for greater
>= (or GEQ)	Test for greater than or equal to

The double special-character representation of NEQ, LEQ and GEQ must be written as shown above. Illegal usage would be =^, =<, or =>.

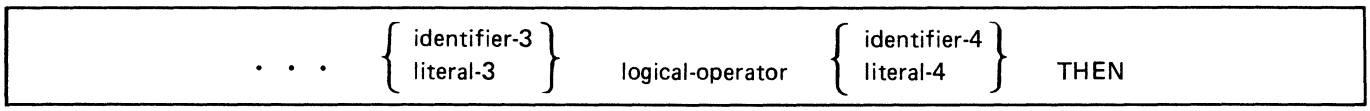


Figure 5-42. Test for Condition-1 with IF, Option 2

An assignment or the result of an arithmetic operation can be used as an operand.

Either operand may be a literal, but not both. A literal, if used, must not exceed two digits (or one byte).

Table 5-8 lists the logical operators (or Boolean operators) that are permitted, in hierarchical order.

Table 5-8. Permitted Logical Operators

Operator	Function
AND	To generate an AND instruction which determines the logical truth of the low-order bits of identifier-3 and identifier-4.If true (both bits are "on"), the THEN action will be taken.If a literal is used, a "bit one test" (BOT) instruction will be generated using the literal as a mask to be generated using the literal as a mask to be repetitively applied against each byte of the entire identifier specified, and if any bit in the mask corresponds to a bit in the identifier, the THEN action will be taken.
OR	To generate an ORR instruction that determines the logical truth of the low-order bits of identifier-3 and identifier-4. If true (either bit or both "on"), the THEN action will be taken. A literal used with the OR operator has no meaning and will generate a syntax error.
NOT (or ^)	To generate a NOT instruction that determines the logical truth of the low order bits of identifier-3 and identifier-4.If true (either bit "on" but not both), the THEN action will be taken. If a literal is used, a "bit zero test" (BZT) instruction will be generated using the literal as a mask to be repetitevely applied against each byte od the entire identifier specified. If any bit in the mask corresponds to a zero (bit off) in the mask corresponds to a zero (bit off) in the identifier, the THEN action will be taken.

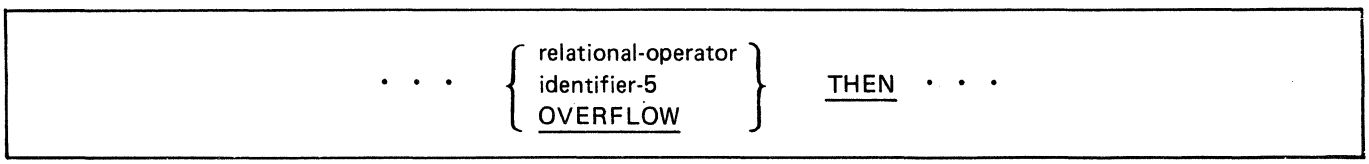


Figure 5-43. Test for Condition-1 with IF, Option 3

If a relational operator is used (see Option 1) a test will be generated against the setting of the comparison indicators. This form is used following, for example, a COMPARE or SCAN command. If the condition is met, the THEN action will be taken.

If an identifier is specified, the low-order bit of the identifier will be tested, and if set (on) the THEN action will be executed. If identifier-5 references a bit, that bit will be tested.

If OVERFLOW is specified, the overflow indicator will be tested. If set (on) it will be turned off (reset) and the THEN action will be executed. To force the overflow indicator off: *IF OVERFLOW THEN:*

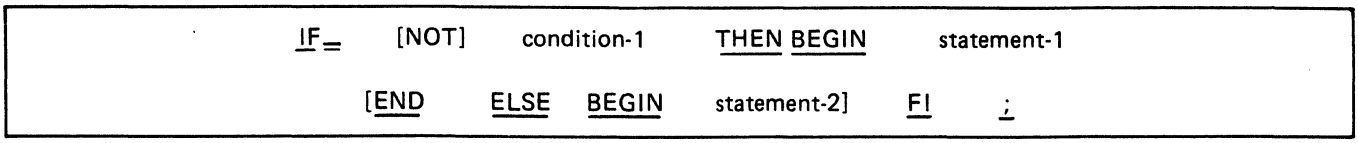


Figure 5-44. Format 2 for the IF Statement

Format 2 includes the features of Format 1, with extensions.

An IF_ statement must end with FI. FI thus provides a visible end to the IF_ statement.

Since FI is an implicit END, THEN must be followed by BEGIN. Because FI is an implicit branch label, the EXITCOND verb may be used within an IF_ statement.

An early exit from an IF_ statement can be caused by EXITCOND. EXITCOND causes control to be transferred to the first executable instruction following FI.

Example:

```
IF_ condition-1  THEN BEGIN
       Y := Y + 1 ;
       IF Y = 10 THEN
          EXITCOND ;        & Bypass the READ
       READ     Z;
FI ;
```

Refer also to ELSE_, ELIF, and THEN_ in Appendix G.

LOCK

The function of LOCK is to lock a block of a shared disk or diskpack file, without the transfer of data taking place. Figure 5-45 describes the format of the LOCK statement.

```
LOCK file-name [WITH SEEK] ;
```

Figure 5-45. Format of LOCK Statement

LOCK file-name locks a block of disk file without transferring data from the file. If the block is currently locked, LOCK waits to relock it. The SEEK option functions the same, excepting it will not wait if the block is currently locked.

While LOCK does not transfer data from the file into the program's buffer, it does destroy the contents of the file's record area. Following a LOCK, therefore: (1) a record must be initialized before a WRITE is executed (2) information previously in the record area may be lost.

Under MCP releases prior to MCP/VS 2.0, the use of LOCK is valid only when the MCP SHRD option is set.

In any case, the FILE declaration must include the SHARED attribute.

OPEN

The function of OPEN is to initiate the processing of both input and output files. The MCP performs checking or writing, or both, of labels and other input-output operations. Figure 5-46 describes the format of the OPEN statement.

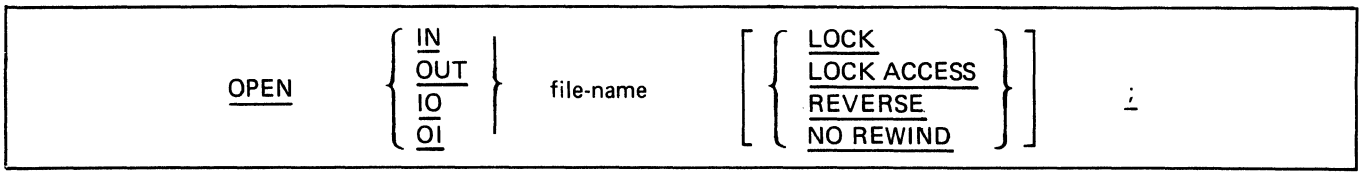


Figure 5-46. Format of OPEN Statement

With every OPEN, the type of OPEN must be specified. Allowable options are IN (input), OUT (output), IO (input-output) or OI (output-input).

The IO and OI options pertain to disk and disk pack files. In addition, IO may be specified for data communication devices.

The file-name must be the name (not identifier) assigned to the file in the FILE declaration.

When an OPEN OUT statement is executed for a magnetic tape file, the MCP searches the assignment table for an available scratch tape, writes the label as specified by the program and executes any label routines for the file. If no scratch tape is available, a message to the operator is typed and the program is suspended until the operator mounts one, or one becomes available due to the termination of a multiprocessing program. OPENing of subsequent reels of multi-reel tape files is handled automatically by the MCP and requires no special consideration from the programmer.

The IO option permits the OPENing of a disk or disk pack file for input and/or output operations. This option assumes the existence of the file on disk, and cannot be used if the file is being initially created. That is, the file to be OPENed must be present in the MCP Disk Directory, or have previously been created and CLOSEd in the same run of the program.

When the IO option is used, the MCP immediately checks the Disk Directory to see if the file-identifier is present, or if this file has been created and CLOSEd in the same run of the program. The system operator will be notified if it is absent, and the file can then be loaded (if available), or the program can be DSed (discontinued).

The OI option is identical to OPEN IO with the exception being that the file is assumed to be a new RANDOM file to the Disk Directory. The OI option does not, nor was it intended to, replace the OPEN IO option, since the use of OPEN OI assumes that a new file is to be created each time.

The LOCK option, executed on a permanent disk or disk pack file, will be performed only on a file not in use by any other program. Once a file is OPENed with LOCK, no other program will be able to OPEN the file until the LOCKing program has CLOSEd it.

The LOCK ACCESS option, executed on a permanent disk or disk pack file, will be performed if the file is not OPEN in any manner other than INput by any other program. Once a file is OPENed with LOCK ACCESS, any program may OPEN the file as INput, but not IO.

With either LOCK or LOCK ACCESS, if the OPEN action cannot be completed by the requesting (or any other) program, that program will be suspended until the program LOCKing the file has CLOSEd; then will be automatically reinstated by the MCP.

The NO REWIND option is used to OPEN magnetic tape files without OPENing (output) the second and all subsequent files on a multi-file reel of magnetic tape.

The REVERSE option can only be used with single reel, single file, tape files. When the REVERSE option is specified, the subsequent READ statements for the file make the data-records available in reverse record order starting with the last record. Each record will be read into its record-area and will appear as if it had been read from a forward moving file.

If the peripheral assigned to the file permits rewind action, the following rules apply:

- When neither the REVERSE nor the NO REWIND option is specified, execution of the OPEN statement for the file will cause the file to be positioned ready to read the first data-record.

- When either the REVERSE or the NO REWIND option is specified, execution of the OPEN statement does not cause the file to be positioned. When the REVERSE option is specified, the file must be positioned at its physical end. When the NO REWIND option is specified, the file must be positioned at its physical beginning.

OVERLAY

The function of OVERLAY is to request the MCP to call in a specified overlayable segment if it is not present and when it becomes present, branch to the first executable instruction in that segment. Figure 5-47 describes the format of the OVERLAY statement.

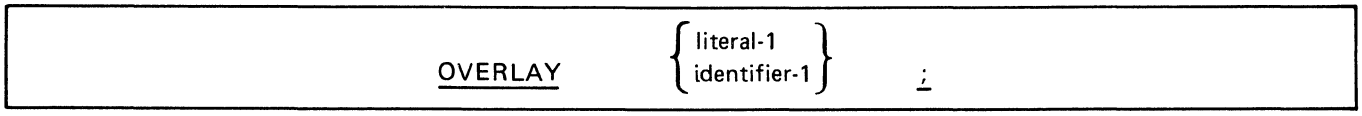


Figure 5-47. Format of OVERLAY Statement

Literal-1 or identifier-1 is the segment dictionary entry for the requested segment. Identifier-1 must be declared INTEGER (3).

If the overlay is not in memory, the MCP reads the requested segment into the appropriate memory area and marks the segment present, thus any future call on a "present" overlay results in a direct branch to the segment without MCP intervention.

Procedure Call

The procedure call statement passes control to (enters) a procedure. After the procedure has been completed, the program control will return (exit) to the statement which follows the calling statement.

The format for a procedure call is:

```
procedure-name    [ (actual-parameter-list) ]    ;
```

Figure 5-48. Format for a Procedure Call

A procedure call is a statement. A procedure call must never appear adjacent to an assignment or comparison operator.

The procedure being called must reside within range. An overlayable procedure may not be called from another procedure occupying the same area of memory. Parameters are optional (see PROCEDURE declaration) but if used must be enclosed in parentheses. Multiple parameters must be separated by commas and may be composed of data-names or literals in any order. Evaluation of the actual parameter list is performed left to right. Only a single "name" or "value" will be passed for each parameter. The actual parameters passed at object run-time will be matched left to right with the formal parameter names contained in the PROCEDURE declaration of the invoked procedure. The maximum number of parameters that can be passed is 10.

A value will not be returned from a called procedure. If such a requirement exists, the result must be communicated through the use of global data-names, or by passing a parameter by name and specifying the corresponding formal parameter in the procedure to the left of a replacement operator within an executable statement.

Table 5-9. Calling Procedures

Examples	Comments
PROX;	The procedure PROX is being invoked.
IF X THEN PROX ELSE PROY;	One of the two procedures will be called depending on the data-name X.

READ

The function of this statement is twofold:

- When processing sequential input files, a READ statement will cause the next sequential record to be moved from an input buffer area to the actual work area, thus making the record available to the program. The READ statement permits a branch to a specified label when an end-of-file condition is detected by the MCP.
- For random file processing, the READ statement communicates with the MCP to explicitly cause the reading of a physical record from a disk file and also allows a branch to a specified label if the contents of the associated KEY data item is found to be invalid.

The format of the READ statement is:

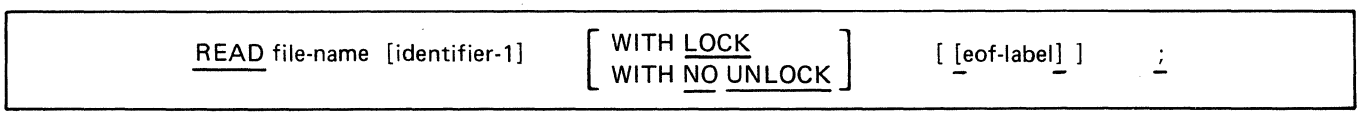


Figure 5-49. Format of READ Statement

An OPEN statement must be executed for a file prior to the execution of the first READ statement for that file.

File-name must be the name (not identifier) of a file declared in a FILE statement.

The use of identifier-1 in a READ statement changes the WORKAREA location (see the FILE declaration) for this and all subsequent READ operations on the specified file. If this option is used, a WORKAREA must have been specified in the FILE declaration.

The eof-label provides an address to which program control will be returned when the logical end-of-file is reached. If used, it must reference a defined label and must be enclosed in brackets. If end-of-file is reached and no eof-label is provided, the program will be terminated.

If the end of a magnetic tape reel is recognized during execution of a READ statement, the following operations are carried out:

1. The standard ending reel label routine and the user's ending reel label routine, if specified by the ROUTINETYPE statement, are carried out.
2. A tape swap is performed.
3. The standard beginning reel label routine and the user's beginning label routine, if specified, are executed.
4. The first data record on the new reel is made available.

For RANDOM files, the READ statement implicitly performs the functions of the SEEK statement. If the contents of the associated KEY data item is out of the range of the file, the MCP will return control to the address specified. For RANDOM files, the sensing of an end-of-file condition does not preclude further READs on that file. For sequential files, a READ following end-of-file is invalid, and the program will terminate.

If a READ parity error occurs, the MCP will retry the READ operation until the record is successfully read, or until a specified number of retry attempts has been reached. If the parity error is unrecoverable, the MCP will branch to the ERROR routine provided by the programmer. If an ERROR routine is not found, the program will be terminated.

The use of the MCP Shared file capability allows three shared disk READ constructs.

Table 5-10. READ Constructs

READ file-name...	READS file even if block is locked.
READ file-name with LOCK	READS and locks a block of a disk file.
READ file-name with NO UNLOCK	Locks a block of a shared file, performs a read, and then unlocks the block. If the requested block is locked by another program, READ WITH NO UNLOCK waits until the record is unlocked. If the block is already locked by the program issuing the READ WITH NO UNLOCK, no delay is necessary. This statement obtains the contents of a record at a time when no other program has the record locked. It differs from READ WITH LOCK in that READ WITH NO UNLOCK does not leave the record locked.

Under MCP releases prior to MCP/VS 2.0, the MCP's SHRD option must be set in order to use the LOCK or UNLOCK options. In any case, the FILE declaration must incLude the SHARED attribute.

SCAN

The function of the SCAN statement is to scan a data field for a delimiting character. Figure 5-50 describes the format of the SCAN statement.

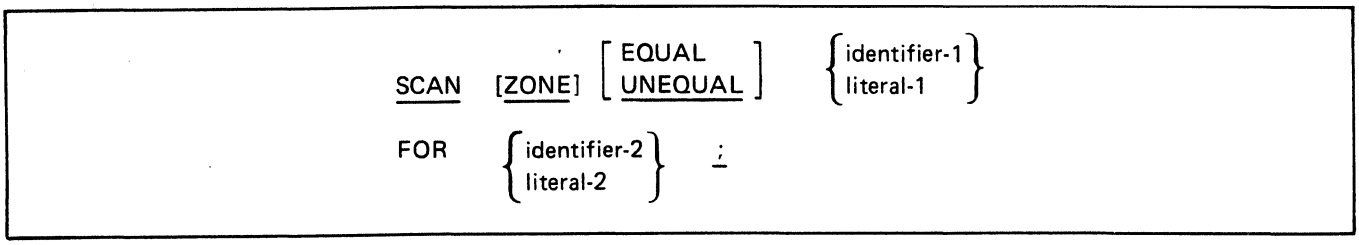


Figure 5-50. Format of SCAN Statement

The SCAN command compares the first identifier-1 field character with all identifier-2 field characters, and if the condition tested for (EQUAL or UNEQUAL) is found, the SCAN is complete. If not, the next identifier-1 field character is compared with all identifier-2 characters, and so forth until a match is found or until the identifier-1 field is exhausted.

Every SCAN instruction stores a character count (not storage position) into Program Reserved Memory location 00038-39 according to the following rules:

- 00 is stored if the first identifier-1 character satisfies the condition tested.
- The number of characters in the identifier-1 field preceding the equal (or unequal) character is stored if the non-first character in the identifier-1 field satisfies the condition tested.
- The length of the identifier-1 field minus one is stored if no identifier-2 field character satisfies the condition tested.

Use of the ZONE and/or the EQUAL/UNEQUAL options permits four variations of the SCAN statement:

1. The SCAN EQUAL option (default):

   If the first character in the identifier-1 field equals any of the delimiters in the identifier-2 field, the comparison is set to LOW. If any character in the identifier-1 field other than the first is equal to one of the delimiters in the identifier-2 field, the comparison indicator is set EQUAL. If none of the characters in the identifier-1 field are equal to any of the delimiters in the identifier-2 field, the comparison indicator is set HIGH.

2. The SCAN UNEQUAL option:

   If the first character in the identifier-1 field is not equal to any of the delimiters in the identifier-2 field, the comparison indicator is set LOW. If any character in the identifier-1 field other than the first is not equal to any of the delimiters in the identifier-2 field, the comparsion indicator is set EQUAL. If all characters in the identifier-1 field are equal to any of the delimiters in the identifier-2 field, the comparison indicator is set HIGH.

3. The SCAN ZONE EQUAL option:

   If the zone portion of the first identifier-1 field character is equal to the zone portion of any of the identifier-2 delimiter-zone characters, the comparison indicator is set LOW. If the zone portion of any character in the identifier-1 field other than the first is equal to the zone portion of any of the identifier-2 field delimiter-zone characters, the comparison indicator is set EQUAL. If no zone portion of any of the identifier-1 field characters is equal to the zone portion of any of the identifier-2 field delimiter-zone characters, the comparison indicator is set HIGH.

4. The SCAN ZONE UNEQUAL option:

   If the zone portion of the first identifier-1 character is not equal to the zone portion of any of the identifier-2 field delimiter-zone characters, the comparison indicator is set to LOW. If the zone portion of any character in the identifier-1 field other than the first is not equal to the zone portion of any of the identifier-2 delimiter-zone characters, the comparison indicator is set EQUAL. If the zone portion of every identifier-1 field character matches an identifier-2 field delimiter-zone character, the comparison indicator is set HIGH.

SEARCH

The function of SEARCH is to cause a search of a table to locate a table element that satisifes a specific condition, and store the address of the table element in IX1.

Figure 5-51 describes the format of the SEARCH statement.

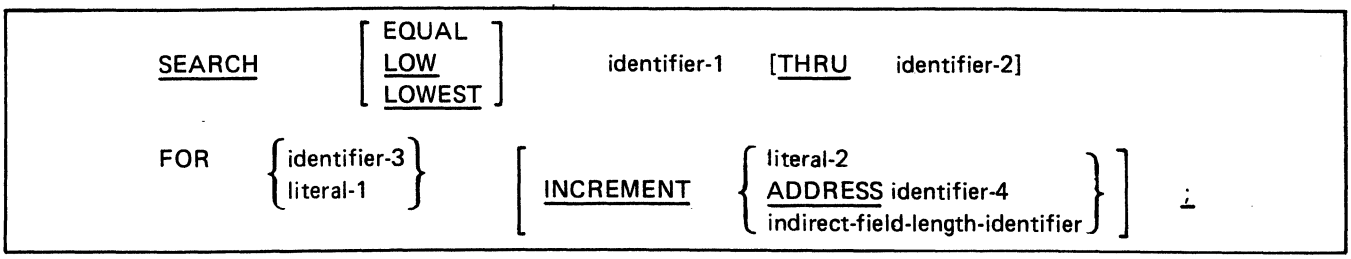


Figure 5-51. Format of the SEARCH Statement

The SEARCH statement will generate a hardware SEARCH command, which uses the C address controller to determine the type of SEARCH: UN indicates SEARCH EQUAL, SN indicates SEARCH LOW, and UA indicates SEARCH LOWEST. If the search type is omitted EQUAL is assumed.

The SEARCH action for each of the three types is as follows:

1. EQUAL (default)

   The value contained in identifier-3 (or literal-1) is compared with the value in identifier-1, then with the value in identifier-1 plus the INCREMENT, and so forth. until an equal condition is detected; or until the address developed by incrementing identifier-1 is equal to or greater than the address of identifier-2 (THRU option). If the THRU option is not specified, the length of the search will be the length defined for identifier-1. If an EQUAL condition is detected, the comparison indicator is set EQUAL, and the address of the EQUAL entry (identifier-1 plus increments) is stored in IX1. If an EQUAL condition is not detected, the comparison indicator is set HIGH and IX1 is unchanged.

2. LOW

   The value contained in identifier-3 (or literal-1) is compared with the value in identifier-1, then the value in identifier-1 plus the INCREMENT, and so forth until an entry is found where the value of identifier-1 (plus increments) is lower than the value of identifier-3; or until the address developed by incrementing identifier-1 is equal to or greater than the address of identifier-2 (THRU option). If the THRU option is not specified, the length of the search will be the length defined for identifier-1. If a LOW condition is detected, the comparison indicator is set EQUAL, and the address of the LOW entry (identifier-1 plus increments) is stored in IX1. If a LOW condition is not detected, the comparison indicator is set HIGH and IX1 unchanged.

3. LOWEST

The value contained in identifier-3 or (literal-1) is compared with the value in identifier-1, then with the value in identifier-1 plus the INCREMENT, and so forth until the address developed by incrementing identifier-1 is equal to or greater than the address of identifier-2 (THRU option). If the THRU option is not specified, the length of the search will be the length defined for identifier-1. If, on any comparison, the value of identifier-1 is lower than the identifier-3 value, that value will be used in all remaining comparisons. If at least one identifier entry is found to be lower than the identifier-3 entry, the comparison indicator is set EQUAL, and IX1 will contain the address of the LOWEST entry found. If no lower entry is found the comparison indicator will be set HIGH, and IX1 will contain the address of identifier-3. If a literal is used, IX1 will contain the address of the literal.

NOTE
The test for the bounds of the SEARCH is done before the SEARCH comparison is made.

The INCREMENT option allows specification of the table entry size. If a literal is used (literal-2), it must be an integer with a value of 1 to 100; and will represent digits or bytes depending on the attributes of identifier-1. If the ADDRESS option is used, a compile-time calculation of the INCREMENT will be performed by subtracting the address of identifier-1 from the address of identifier-4. The difference must fall within the value range defined for literal-2. If the INCREMENT entry is omitted, the default value (of literal-2) will be one.

When indirect addressing is used on identifier-2, it is the programmer's responsibility to set the appropriate address controller in the final address field.

An indirect address may not be used for identifier-1 or identifier-4 if the ADDRESS option is used. A syntax error will result because the indirect address does not provide sufficient information for calculation of the increment.

Because comparison length is independent of the entry length, the SEARCH statement may be used for scanning as well as table lookup. This is accomplished by addressing the data area to be scanned (identifier-1 THRU identifier-2), the keyword address (identifier-3) and an INCREMENT value of one (1). When this occurs, the entire data field will be scanned for occurrence of the keyword.

SEARCH LINK/DELINK

The function of the SEARCH LINK statement is to search a non-contiguous table for an element that satisifes a specific condition, and store the address of the table element in IX1. SEARCH DELINK performs the same function but in addition saves the address of the previous table entry.

Figure 5-52 shows the format of the SEARCH LINK DELINK statement.

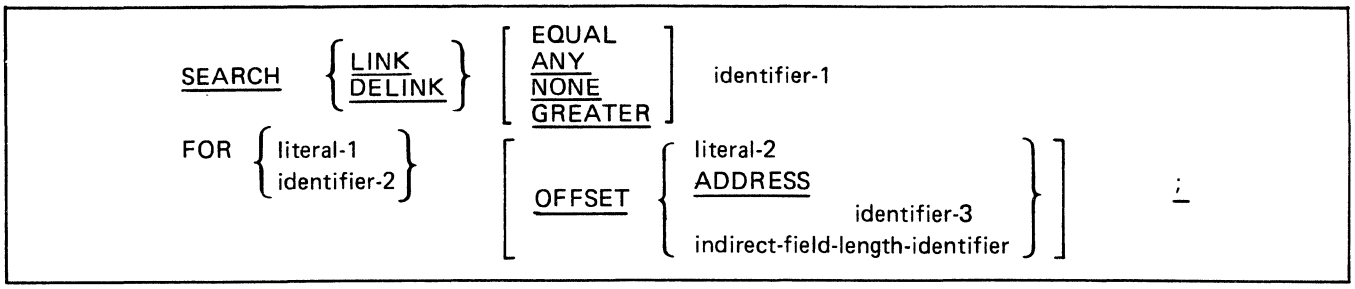


Figure 5-52. Format of the SEARCH LINK DELINK Statement

The SEARCH LINK (or DELINK) instruction will test the table element addressed by identifier-1 in the manner prescribed by the search mode (described below). If the condition is not met, the first six digits contained in this table element (called link address) will be used as the address of the next table element to be tested, etc. until the condition is met, or until the first six digits of an element are zeros. If the condition tested for is found, the address of the table element will be placed in IX1. In addition, if DELINK is specified, the address of the previous table element is placed in IX2. If the condition is not met, IX1 and IX2 are not changed.

Either LINK or DELINK must be specified. The search mode defines the type of SEARCH to be performed. If the search mode is not specified, EQUAL is assumed. Allowable options are:

- EQUAL (default)

  The contents of each table element are compared with the contents of identifier-2, or with literal-1 until an equal condition is found. If an equal condition is found, the comparison indicator is set EQUAL, otherwise it is set HIGH.

- ANY

  The contents of identifier-2 or literal-1 are a mask which specify bits of the table elements to be considered in the test. A bit on in the mask signifies that the corresponding bit in the table element is to be considered; a bit off in the mask signifies that the corresponding bit in the table element is to be ignored. If any of the bits considered in the table element is on, the search is satisfied. If such a match is found, the comparison indicator is set EQUAL; otherwise it is set HIGH.

- NONE

  The contents of identifier-2 or literal-1 are a mask which specify bits of the table elements to be considered in the test. A bit on in the mask signifies that the corresponding bit in the table element is to be considered; a bit off in the mask signifies that the corresponding bit in the table element is to be ignored. If all the bits considered in the table element are off, the search is atisfied. If such a match is found, the comparison indicator is set EQUAL; otherwise it is set HIGH.

- GREATER

  The contents of each table element are compared with the contents of identifier-2, or literal-1 until an elment is found that is less than or equal to the contents of identifier-2. If none are found, the comparison indicator is set HIGH. If an EQUAL condition is found, the comparison indicator is set EQUAL; if a less than condition is found, the comparison indicator is set LOW.

The address controller of identifier-1 is set by the type of SEARCH requested, as follows:

0 - EQUAL
1 - ANY
2 - GREATER
3 - NONE

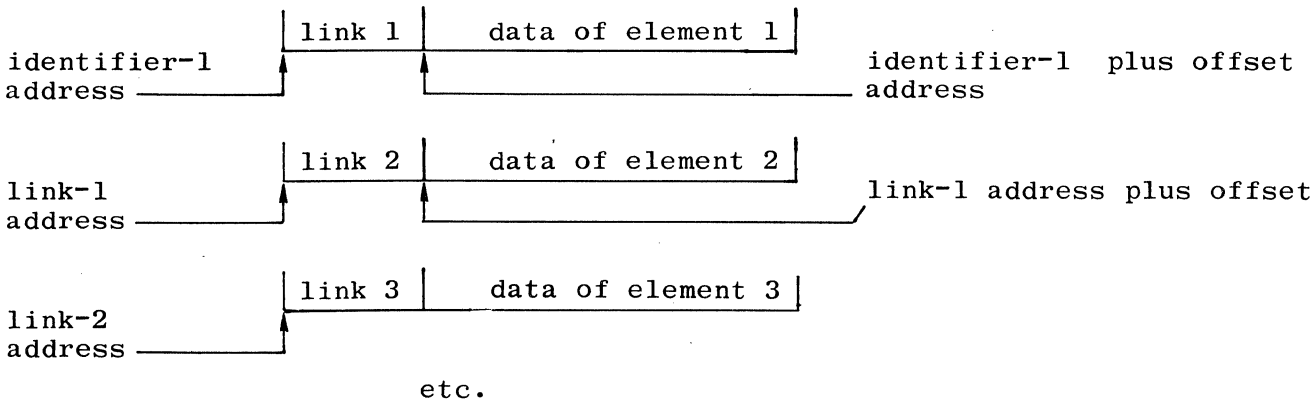
Indirect addressing cannot be specified for identifier-1. Indexing may be used with identifier-1, however neither indirect addressing nor indexing may be used on any link address.

The address controller of identifier-2, or literal type, if literal-1 is used determines the data type of both fields; identifier-1 and identifier-2. The length of the field to be tested is determined by the length of identifier-2, and must not exceed 100 (digits or bytes). Indexing may be used with identifier-2; indirect addressing may be specified only if CONTROL OP B4700 is used.

The OFFSET option defines the location in the table element to be used for comparison. If not specified, the default value is zero, resulting in a test on the link address. If a literal is used, that value, digits or bytes, depending on the address controller specified for identifier-2, is added to identifier-1 to determine the starting point in the table element for the comparison. Again the length of the field to be compared is determined by the length of identifier-2. If the ADDRESS option is used, identifier-3 will address the field in the table element to be used and the compiler will calculate the OFFSET value.

NOTE
If CONTROL OP B4700 has been specified, a Search Link List or Search Link Delink instruction will be generated to perform this function.

Typically, the table would appear as:



In the three-element table shown in the example above, if the conditions were met while pointing to element 3 during a SEARCH LINK statement, IX1 would contain the address in link-2 (that is, the "link-2 address" which is the address of element 3). If the conditions were met during a SEARCH DELINK statement, IX1 would point to link-2 address and IX2 would point to link-1 address.

SEEK

The function of SEEK is to initiate the accessing of a disk or disk pack record for a subsequent READ and/or WRITE operation.

Figure 5-53 shows the format of the SEEK statement.

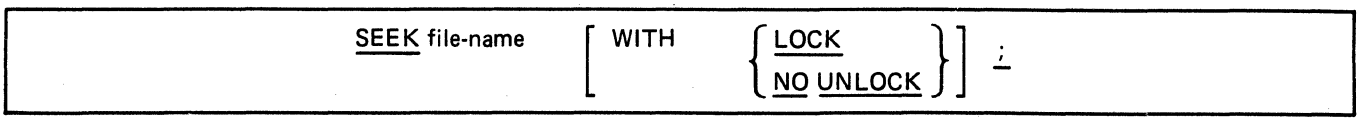


Figure 5-53. Format of the SEEK Statement

A SEEK statement pertains only to disk or disk pack storage files in the RANDOM access mode, and may be executed prior to the execution of each READ and/or WRITE statement.

The SEEK statement uses the contents of the data-name in the associated KEY clause for the location of the record to be accessed. At the time of execution, the determination is made as to the validity of the contents of the KEY data item for the particular disk storage file. If the key is invalid, the invalid key branch ([eof label]) of the next executed READ or WRITE statement for the associated file is taken.

The key identifier should be set to the desired record before the SEEK is initiated. To preclude the possibility of overlaying input buffers, more than one data area should be specified in the FILE declaration.

Two or more successive SEEK statements for a random storage file may logically follow each other. Any validity check associated with the first SEEK statement is negated by the execution of a second explicit or implied SEEK statement.

If a READ or WRITE statement for a file assigned to disk or disk pack is executed, but an explicit SEEK has not been executed since the last previous READ or WRITE for the file, the implied SEEK statement is executed as the first step of the READ/WRITE statement.

An explicit alteration of KEY after the execution of an explicit SEEK has been performed, but prior to a READ/WRITE, causes the initiation of an implied SEEK of the specified record, negating the value of the explicit SEEK.

The use of the MCP Shared file capability allows three shared disk SEEK constructs.

Table 5-11. Shared Disk SEEK Constructs

SEEK file-name	Seeks even if locked.
SEEK file-name with LOCK	Seeks record and locks a block of a disk file.
SEEK file-name with NO UNLOCK	Initiates a LOCK and a SEEK. If the requested block is locked by another program, SEEK NO UNLOCK waits until the record is unlocked. If the block is already locked by the program issuing the SEEK NO UNLOCK, no delay is necessary. This statement obtains the contents of a record at a time when no other program has the record locked. It differs from SEEK LOCK in that SEEK NO UNLOCK does not leave the record locked.

Under MCP releases prior to MCP/VS 2.0, the MCP's SHRD option must be set in order to use the LOCK or UNLOCK options. In any case, the FILE declaration must include the SHARED attribute.

SORT

The function of the SORT statement is to perform a disk sort on an input file of records by transferring such data into a disk work file and sorting those records on a set of specified keys. The final phase of the sort operation produces an output file in the specified sequence.

The SORT statement invokes the MCP's SORT. intrinsic.

Figure 5-54 shows the format of the SORT statement.

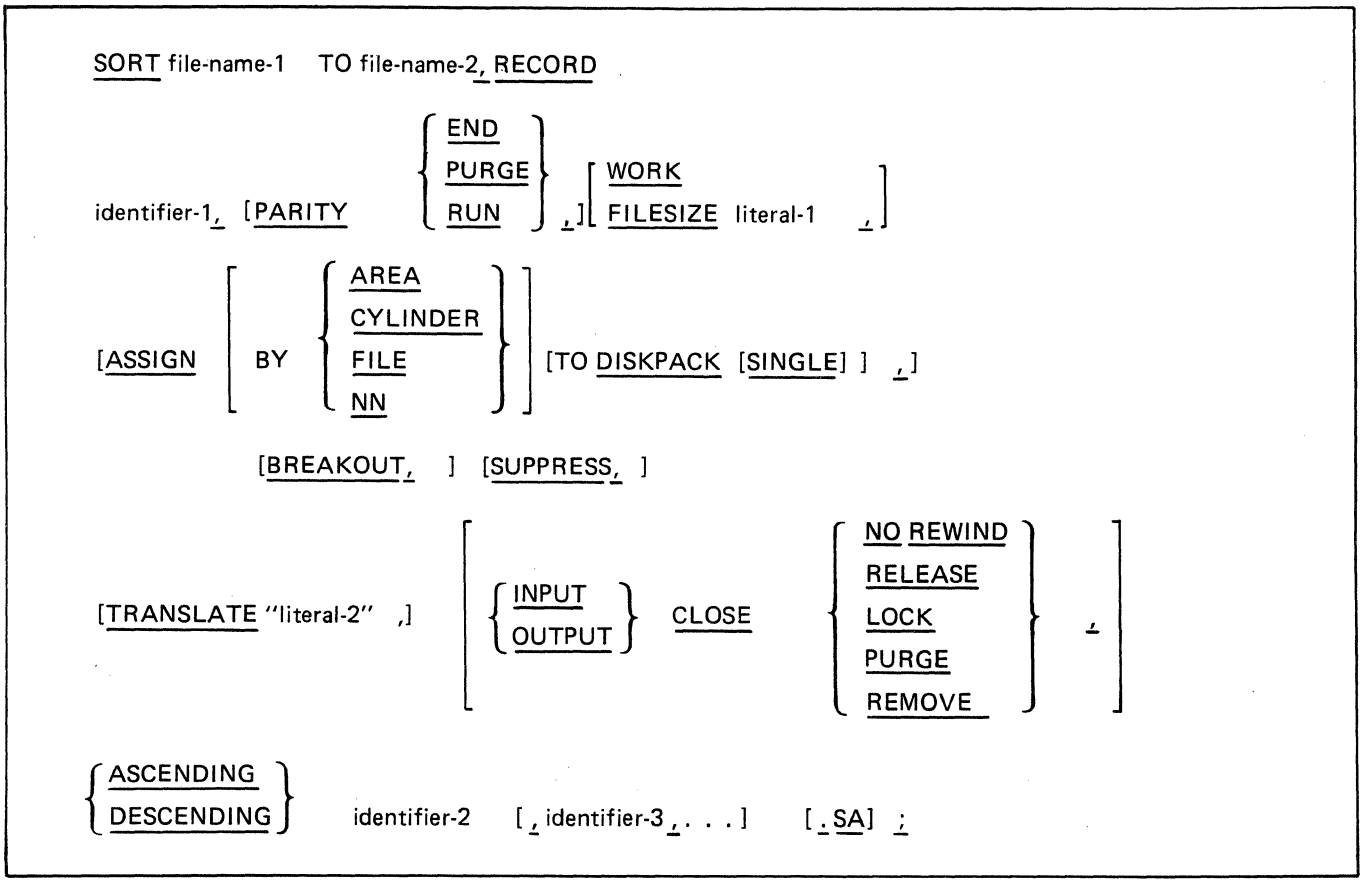


Figure 5-54. Format of the SORT Statement

The SORT option should be specified in the file statement of the file to be sorted (file-name-1) to ensure that sufficient memory is allocated for execution. All records residing in file-name-1 will be transferred to file-name-2 in the specified sequence upon encountering the generated SORT verb object code. At the time of execution of the SORT statement, file-name-1 must not be OPEN. The SORT statement automatically performs the function necessary to OPEN, READ, and CLOSE file-name-1.

Also, at the time of execution of the SORT statement, file-name-2 must not be OPEN. File-name-2 will be automatically OPENed before the sorted records are transferred from the work file and in turn, will be CLOSEd after the last record in the work file has been transferred.

Identifier-1 is the name of the sort RECORD area and will be used for the declaration of the sort keys; identifier-2, identifier-3, and so forth.

The PARITY option, if used, may specify one of the following actions to be performed on an irrecoverable I/O parity error:

- END (default)

  The program will be terminated with appropriate operator notification.

- PURGE

  All records in the block containing the parity error will be dropped.

- RUN

  All records in the block containing the parity error will be retained, and sorted as if the parity error had not occurred.

The WORK option is used to specify the number of records per area. The FILESIZE option is used to specify the total number of records in the file to be sorted. Literal-1 may be up to 8 digits. For DISK or DISKPACK input files this clause is not used, but may be specified for documentation. For input files other than DISK or DISKPACK the clause should be included for more efficient sort performance.

The ASSIGN option is used to specify work file assignment technique, and the allowable options have the same meaning as described for the FILE declaration.

DISKPACK is used to indicate that disk packs are to be the work file medium; SINGLE restricts the sort to a single disk pack.

BREAKOUT specifies that rerun points will be available during the SORT.

SUPPRESS specifies that the record count will not be printed on the SPO.

The TRANSLATE clause may be used if a collating sequence other than the standard hardware collating sequence is required. Literal-2 is a 1-6 character literal used by the sort as the file-id of the translation table. This must be 400 byte single area file on head-per-track disk. The collating sequence specified in the translate file replaces the normal hardware collating for the sort key fields described as unsigned 8 bit (alpha-numeric). If there are 4 bit fields in the sort key, the translation will not apply to those parts of the sort key. Translate table files in the format required by the sort, may be created through the use of the MAKTRN program.

The CLOSE clause is optional and may be used to specify the type of CLOSE required on file-name-1 (input) and file-name-2 (output).

ASCENDING and DESCENDING specify the direction of the sort on each key. The sort key can be mixed ascending and descending, if desired, but their total length cannot exceed 290 bytes. Those may be up to 40 individual sort keys subject to the foregoing length restriction.

The ".SA" option may be used as an override on the sort keys to indicate signed alpha.

Refer to the System Software Operation Guide (SOG) for a discussion of MAKTRN and the SORT. intrinsic: Volume 2 in MCP/VS 1.0 and MCPIX SOGs, and Volume 3 in MCP/VS 2.0 or later SOGs.

SORT RETURN

The function of the SORT RETURN is to request a return from the sort intrinsic to the user program. Figure 5-55 shows the format of the SORT RETURN statement.

```
SORT RETURN ;
```

Figure 5-55. Format of the SORT RETURN Statement

The SORT RETURN statement is used only by the SORT intrinsic.

SPACE

The statement is used to cause forward spacing of line printer paper, or forward/reverse record spacing on magnetic tape, paper tape, head-per-track disk or disk pack files.

Figure 5-56 shows the format of the SPACE statement.

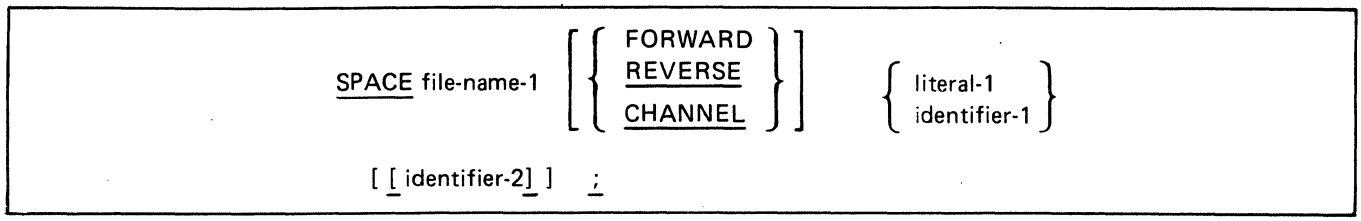


Figure 5-56. Format of the SPACE Statement

File-name-1 may only be assigned to TAPE, DISK, PRINTER, PTREADER, or DISKPACK. Literal-1 must be numeric.

For non-printer files identifier-1 or literal-1 represents the number of records to be spaced (or positioned) and should not exceed 4 digits in length. For a printer file, identifier-1 or literal-1 represents either the number of lines to be spaced, or the channel number (on a carriage control tape), neither of which should exceed 2 digits in length.

Identifier-2 specifies the end-of-file or end-of page label, and must be enclosed in brackets (indicating address constant).

Space Construct:

REVERSE is an illegal option if a print file is specified.
CHANNEL is an illegal option if a non print file is specified.
Output paper tape files may not be saved.
Output magnetic tape files may be REVERSE spaced only.

SPOMESSAGE

The functions of the SPOMESSAGE statement are to pass keyboard input messages to the MCP and to request that responses be returned to the program.

Figure 5-57 describes the format of the SPOMESSAGE statement.

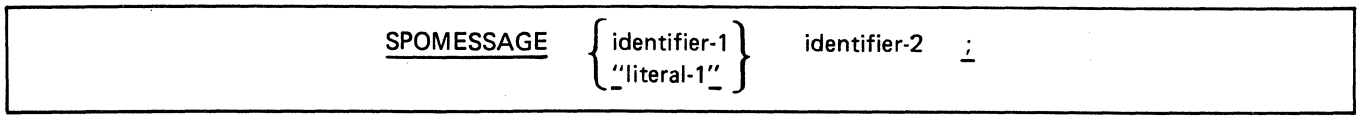


Figure 5-57. Format of the SPOMESSAGE Statement

Identifier-1 or literal-1 may be a data-name of an ALPHA keyboard input message or a non-numeric literal being the keyboard input message itself. This message must be in the same format as the keyboard input message actually typed at the ODT, and should be terminated by either a period or an ETX character (%03%).

The following keyboard commands may be passed to the MCP in identifier-1 or literal-1 of the SPOMESSAGE statement.

Table 5-12. Keyboard Commands in SPOMESSAGE

AJ	CN	FN	MR	RA	ST	WM
AX	DA	FP	MX	RB	SW	WOWQ
BD	DB	FR	NL	RD	TI	WQWS
BF	DC	GO	NT	RF	TO	WSWT
BK	DM	GT	OF	RK	UL	WTWX
BP	DP	HN	OK	RM	UP	WX
BR	DQ	IL	OL	RS	WB	WXD
CA	DS	IN	OT	SB	WC	WXM
CD	FA	LC	OU	SK	WD	WXP
CK	FM	LP	PD	SS	WJ	WY
						XC

If a keyboard command other than those listed in the preceding text is passed, identifier-2 contains @0707@ followed by **KBD IGNORED: REQUEST NOT ALLOWED.

Each response line is placed in identifier-2 as it would appear on the SPO (including removal of extraneous blanks if applicable). Each line is terminated by carriage return and line feed characters (%DOA%;) the last (or only) line is additionally terminated by an ETX character (%03%).

Identifier-2 must be at least 160 digits long. If identifier-2 is too small for all lines of the response, a NULL character (%00%) follows the full last line which could fit into the area. If no lines could fit the first character of identifier-2 is NULL.

Identifier-1 and identifier-2 must not share any portion of their memory allocation.

STOP

The function of the STOP statement is to halt the object program temporarily or to terminate execution.

Figure 5-58 shows the format of the STOP statement.

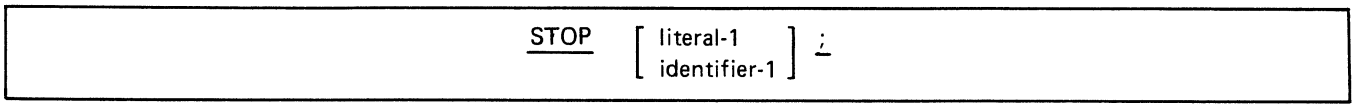


Figure 5-58. Format of the STOP Statement

If STOP is used alone, then all files which remain OPEN will be CLOSEd automatically. Output files assigned to DISK or DISKPACK, when FILE declarations do not include the SAVE FILE option, will be CLOSEd PURGE and all others will be CLOSEd RELEASE. All storage areas for the object program are returned to the MCP and the job is then removed from the MCP Mix.

The STOP is not used for temporary stops within a program. STOP must be the last statement of the program execution sequence.

If the literal-1 or identifier-1 option is used, it will be DISPLAYed on the SPO and the program will be suspended. When the operator enters the MCP continuation message mix-index AX, program execution resumes with the next sequential operation. This option is normally used for operational halts to cause the system's operator to physically accomplish an external action.

STOQUE

The function of the STOQUE type statements is to receive/send data from/to a storage queue via the MCP STOQUE mechanism.

Figure 5-59 shows the format of the STOQUE type statement.

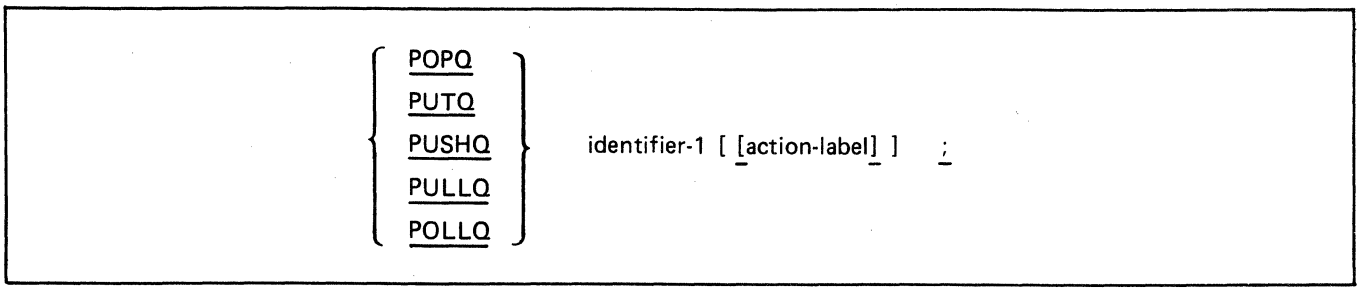


Figure 5-59. Format for STOQUE Statements

Identifier-1 contains the address of the program's Storage Queue Parameter Block (see following text).

Action-label is not valid with POLLQ, and is optional for the other constructs. When present, it specifies the label to which the program will branch if either of the following conditions occur and the program is not to wait for resolution: a) storage space is not available for a storage request; b) the data sought in a retrieval request is not in the queue. These conditions are further explained in the following paragraphs.

The Storage Queue (STOQUE) functions are performed by the STOQUE extension module of the MCP. If its facilities are used, it must be loaded into memory by setting the STOQ system option. Under the 2.0 and later releases of MCP/VS, STOQ is loaded automatically when it is needed.

The STOQUE module has the basic function of transferring data from a data area in the program to an external memory buffer and retrieving that data upon request. The mechanism may be used simultaneously by any number of programs as a means to transfer data between processes, or even as temporary storage for a single process. The data elements placed into the memory buffer are organized into one or more program-independent, symbolically named lists called storage queues.

The STOQUE mechanism differs significantly from the Core to Core Feature (see the FILL verb) in that no synchronization of the sending and receiving programs is required. This is due to the fact that STOQUE does not transfer data directly from one program to another but stores the message in an external memory area until it is requested. This means that multiple transactions may be in the storage queue simultaneously; thus the use of STOQUE permits the complete overlap of processing between programs, with no necessity to interlock for each transaction.

All requests made of the STOQUE function refer to the STOQUE Parameter Block area in the user program, whose address is given in identifier-1 preceding. This program-maintained area contains the information needed by STOQUE to control the data elements; its format is as follows:

Table 5-13. Storage Queue Parameter Block

ALPHA <queue name>	(6)	& Identifies Individual & Queue
INTEGER <entry name length>	(2)	& Name Length in Bytes & (0=NULL)
ALPHA <entry name>	(NN)	& Entry Name (Optional)
INTEGER <entry data length>	(4)	& Data Length in Bytes & (0=NULL) & POLLQ Response Area
ALPHA <entry data>	(NNNN)	& Data Field (Optional)

Notes

1. <queue name> identifies the programmatically assigned symolic name of the queue list to which the request pertains.

2. <entry name length> specifies the size of the optional entry name field.

3. <entry name>, if present, specifies the name associated with the individual queue entry. This name may be used to provide a substructure to a list and provides the means to access data elements which are at locations other than the top or bottom of the queue.

4. <entry data length> specifies the size of the entry data area which in turn contains the transaction to be accessed for a storage request. The maximum size of the entry data field is 2300 bytes. This field serves as the response area for a queue inquiry request (POLLQ).

5. <entry data>, if present, contains the data to be added to the queue in a PUSHQ or PUTQ operation; it receives the data in a POPQ or PULLQ operation. This field is not applicable to a queue inquiry request (POLLQ) operation.

Programs may execute three types of calls on the STOQUE module: store data, retrieve data and queue inquiry.

The storage constructs, PUSHQ and PUTQ, put data into the queue at the top or bottom, respectively. The first PUSHQ or PUTQ executed cause the creation of a queue with the name specified if it does not already exist. If insufficient space is available in the queue for the storage request, the sending program is suspended until space becomes available unless the optional action-label (above) has been specified.

The retrieval constructs, POPQ and PULLQ, retrieve data from the top or bottom of the queue respectively. If the designated queue is empty or no individual entry satisfies any specified name constraint (see following), the program is suspended until the desired element is placed into the queue unless the optional action-label (preceding) has been specified.

If the queue element being retrieved contains more than <entry data length> characters, only the first <entry data length> characters are returned. If the queue element contains fewer characters, <entry data length> is adjusted to show the number of characters actually returned.

The storage and retrieval functions have important variations:

- Elements may be added to or removed from a list at either the top or bottom, yielding the benefits of both FIFO (first-in-first-out) and LIFO (last-in-first-out) mechanisms.

- The elements in a queue list may be named, using <entry name> and <entry name length> above. Access in that case is first to the queue named in <queue name>, then to the specific element named in <entry name>.

  The <entry name> need not be unique; several elements in the queue may have the same name. The name given to a queue element when stored may be longer than the <entry name> specified in a retrieval request: the queue is then searched for an element whose name, in the first <entry name length> characters, matches the specified <entry name> mask. In either of these cases of duplicate element names in a queue, the element accessed is the first matching name from the top of the queue on a POPQ or the first match from the bottom of the queue on a PULLQ.

The queue inquiry function provides a rapid means of determining the size of a list without disturbing any elements.

The inquiry construct, POLLQ, returns a count of the number of entries in a queue as an unsigned integer in the <entry data length> field. If an <entry name> is specified, the count is the number of entries for that name or name group only. If the <entry name> is omitted, the count is the total number of entries in the queue. A response of zero means that the queue or the designated portion of the queue is empty.

STORE

The function of STORE is to produce a string of constants into the object program.

Figure 5-60 shows the format of the STORE statement.

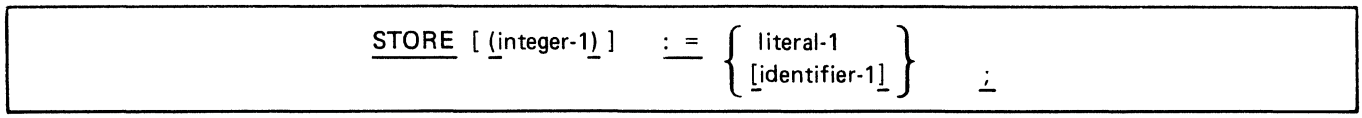


Figure 5-60. Format of the STORE Statement

The STORE statement inserts a specified constant into the program. The constant is included in-line in the code string. Any hardware instruction may be built using this construct.

All presetting values are preset left-justified in the in-line constant.

Integer-1 is optional. If present, it indicates the number of digits or characters of constant. Digits are assumed unless the presetting value indicates alpha.

Identifier-1, if present, must be enclosed in brackets. The constant stored is the ADDRESS OF identifier-1. Literal-1 may be either a numeric literal, or an alpha literal enclosed in quotes.

If identifier-1 or literal-1 are not present, the STORE statement is used only to allocate space.

Subroutine Call

The subroutine call statement passes control to a Type I ICM subroutine from within another Type I ICM. After the Type I ICM has been completed, program control will return to the statement following the calling statement.

Figure 5-61 describes the format of a subroutine call.

```
subroutine-name [ (actual-parameter-list) ]  ;
```

Figure 5-61. Format of a Subroutine Call

Parameters are optional (see SUBROUTINE declaration), but if used, must be enclosed within parentheses. Multiple parameters must be separated by commas, and may be comprised of data-names or literals.

The maximum number of parameters that can be passed is 10.

The procedure call statement explanation contains much information useful in subroutine calls.

The maximum number of subroutine call statements in a Type I ICM is 100.

TOPLOOP

The TOPLOOP statement causes an immediate branch to the first statement within a DO_ loop. Figure 5-62 shows the format of the TOPLOOP statement.

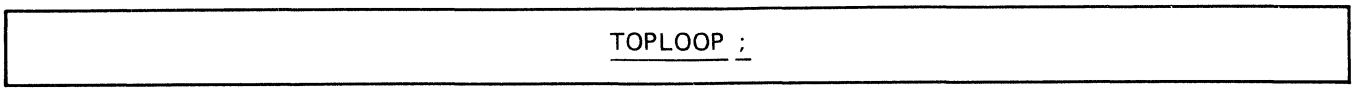


Figure 5-62. Format of the TOPLOOP Statement

TOPLOOP is permitted only within the DO_ version of the DO statement. When executed, it causes control to be transferred to the first statement following DO_ (statement-1 in the DO_ format illustration).

Refer to DO for further details.

TRACE

The function of TRACE is to create documentation of all normal mode processing events and to output this data on a line printer.

Figure 5-63 describes the format of the TRACE statement.

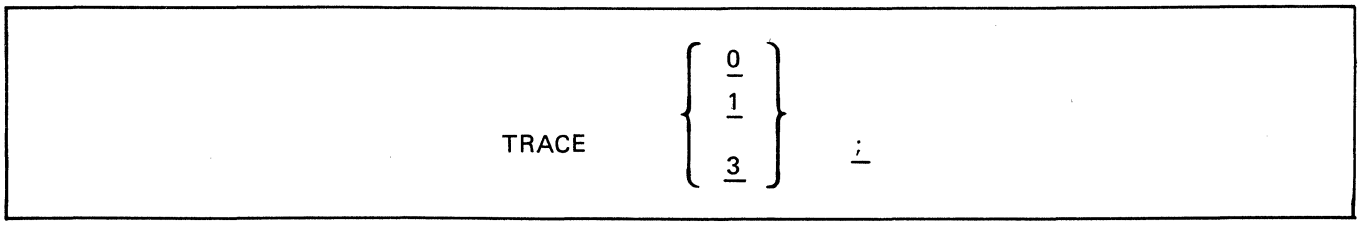


Figure 5-63. Format of the TRACE Statement

When a TRACE statement is encountered during object program execution, one of the following actions will take place at that point in the program:

- The 0 or FALSE option will turn the TRACE off.
- The 1 or TRUE option will cause a TRACE to an available printer of all normal mode instructions until such time as any of the other options are encountered.
- The 3 option will cause a TRACE to backup disk of all normal mode instructions until such time as any of the other options are encountered.

TRANSLATE

The TRANSLATE statement is used to translate a string of digits/characters according to a specified equivalence table and to store the translated string in a specified location.

Figure 5-64 describes the format of the TRANSLATE statement.

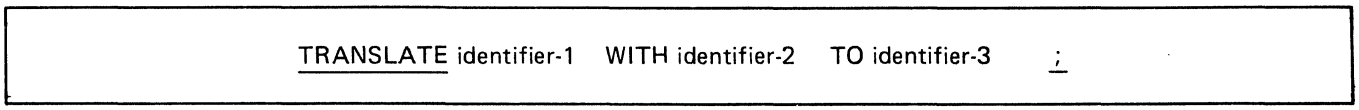


Figure 5-64. Format of the TRANSLATE Statement

Identifier-1 has a maximum size of 10,000 characters/digits. If identifier-1 is signed, the sign will be ignored. If it is integer, the digits will be treated as characters by assuming numeric zone digits.

Identifier-2 must be at Modulo 1000 address. This may be accomplished by the MOD 1000 statement in the declaration. This is the equivalence table.

Identifier-3 is the field into which the TRANSLATEd data will be stored.

Its type must be 8-bit (ALPHA or NUMERIC).

NOTE

The size of the string is calculated as being the smaller of the sending (identifier-1) or receiving (identifier-3) fields. To use indirect field lengths, indirect-field-length-ids should be placed in the controllers of identifier-1 (AF) and identifier-2 (BF). Care should be exercised, in that using only one of the indirect field lengths (AF or BF) may yield unpredictable results.

The character translation is performed by the following algorithm. Refer to Figures 5-65, 5-66, and 5-67, and to Table 5-14.

1. A low-order zero bit is appended to the 8-bit character to be translated.
2. The resultant nine bits are divided into three 3-bit groups.
3. The 3-bit groups are interpreted as three octal digits.
4. The three octal digits are ORed with the address of identifier-2 to produce the hundreds, tens, and units positions of a table-access address.
5. The character at the derived table-access address is moved to a position in identifier-3 corresponding to the original character's position in identifier-1.

NOTE

Since the address generated in the preceding steps is ORed with the low-order 3 digits of the translation table address from the instruction, any one-bits present in this portion of the address will remain. This could produce an invalid address containing undigits. Therefore, the translation table address must be modulo 1000.

Example:

EBCDIC A is the binary number 11000001_2

The stages of translation for this example are (the steps referred to in this list come from the list preceding the NOTE, above):

a. If step 1, the binary number 11000001_2 becomes 110000010.

b. If step 2, 110000010 becomes 110 000 010.

c. If step 3, 110 00010 becomes the octal number 602_8.

d. If step 4, table access address B + 602 is generated.

e. If step 5, the character in B + 602 is moved to the C field.

Figure 5-65 shows the general layout of translation tables in memory. Crossed out areas in figure 5-65 are not accessed by the TRANSLATE instruction.

Following Table 5-14 is a sample BPL program which illustrates the coding and use of a translation table.

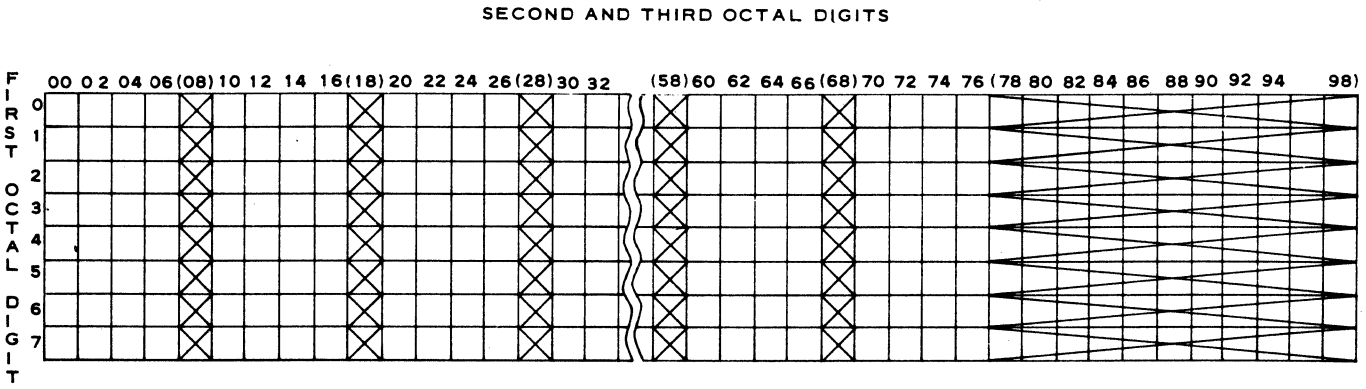


Figure 5-65. Translate Tables in Memory

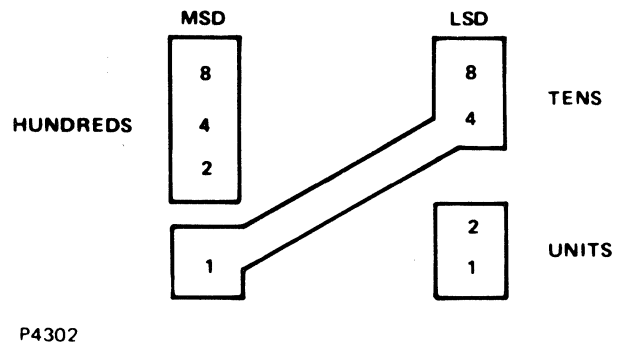


Figure 5-66. B Address (Identifier-1) Modification

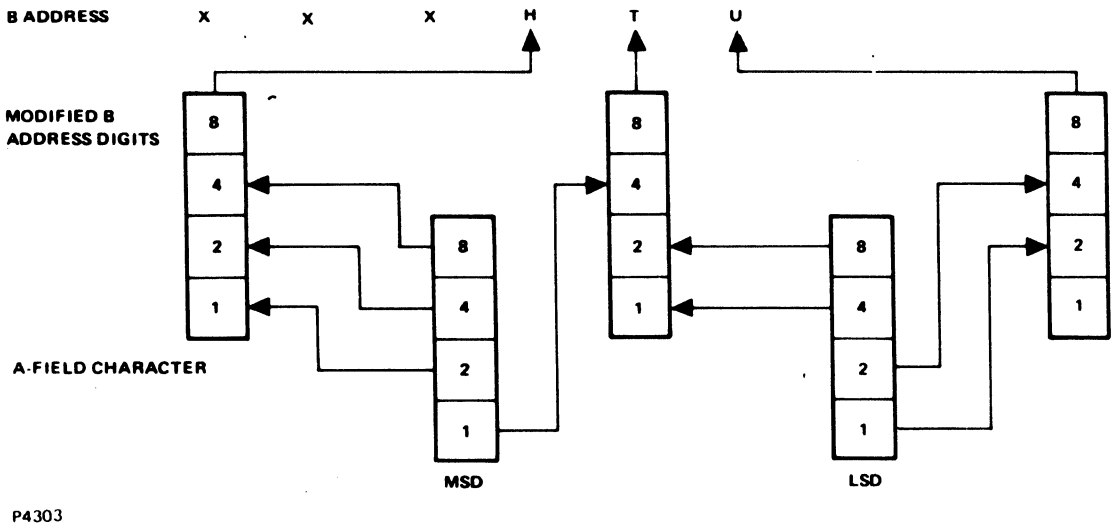


Figure 5-67. B Address (Identifier-2) Modification

Table 5-14. Translate Table Address

A-Character	B INDEX		
	H*	T	U
1F	0	7	6
1E	0	7	4
1D	0	7	2
1C	0	7	0
1B	0	6	6
1A	0	6	4
19	0	6	2
18	0	6	0
17	0	5	6
16	0	5	4
15	0	5	2
14	0	5	0
13	0	4	6
12	0	4	4
11	0	4	2
10	0	4	0
0F	0	3	6
0E	0	3	4
0D	0	3	2
0C	0	3	0
0B	0	2	6
0A	0	24	
09	0	2	2
08	0	2	0
07	0	1	6
06	0	1	4
05	0	1	2
04	0	1	0
03	0	0	6
02	0	0	4
01	0	0	2
00	0	0	0

* Hundreds values are developed with the following A-character zone digit values:

A-Character Zone	H Value
0 + 1	0
2 + 3	1
4 + 5	2
6 + 7	3
8 + 9	4
A + B	5
C + D	6
E + F	7

BPL Program with Translation Table

```
BEGIN
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&  Demonstrate the TRANSLATE statement and a translation
&  table.
&  A translate table in memory has eight 100-digit entries,
&  each holding two 40-digit fields and a 20-digit filler.
&  Each 40-digit (20-byte) field has 16 useable positions,
&  in four groups of four useable positions and a filler.
&  There is, therefore, one useable position in the table for
&  each of the 256 possible characters in the order 00 to FF.
&  To translate, place the new character in the table position
&  of the old character.
&  For example, this table translates lower case letter "a"
&  (81) to letter "A" (C1) by placing the code C1 at the 81
&  position.
&  All lower-case letters (81-89, 91-99, A2-A9) are similarly
&  replaced by C1-C9, D1-D9, E2-E9.
&  To leave a character untranslated, put its own code at that
&  code's position in the table.
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

    ALPHA
     DATA_IN  (40),
     DATA_OUT  (40);
    ALPHA TRANSLATE_TBL (400) MOD 1000;
     ADDRESS = TRANSLATE_TBL;
     INTEGER
    (40) := @000102030004050607000809 0A0B000C0D0E0F00@, & 00-0F
    (40) := @101112130014151617001819 1A1B001C1D1E1F00@, & 10-1F
    (20),
    (40) := @202122230024252627002829 2A2B002C2D2E2F00@, & 20-2F
    (40) := @303132320034353637003839 3A3B003C3D3E3F00@, & 30-3F
    (20),
    (40) := @404142430044454647004849 4A4B004C4D4E4F00@, & 40-4F
    (40) := @505152530054555657005859 5A5B005C5D5E5F00@, & 50-5F
    (20),
    (40) := @606162630064656667006869 6A6B006C6D6E6F00@, & 60-6F
    (40) := @707172730074757677007879 7A7B007C7D7E7F00@, & 70-7F
    (20),
    (40) := @80C1C2C300C4C5C6C700C8C9 8A8B008C8D8E8F00@, & 80-8F
    (40) := @90D1D2D300D4D5D6D700D8D9 9A9B009C9D9E9F00@, & 90-9F
    (20),
    (40) := @A0A1E2E300E4E5E6E700E8E9 AAAB00ACADAEAF00@, & A0-AF
    (40) := @B0B1B2B200B4B5B6B700B8B9 BABB00BCBDBEBF00@, & B0-BF
    (20),
    (40) := @C0C1C2C300C4C5C6C700C8C9 CACB00CCCDCECF00@, & C0-CF
    (40) := @D0D1D2D300D4D5D6D700D8D9 DADB00DCDDDEDF00@, & D0-DF
    (20),
    (40) := @E0E1E2E300E4E5E6E700E8E9 EAEB00ECEDEEEF00@, & E0-EF
    (40) := @F0F1F2F300F4F5F6F700F8F9 FAFB00FCFDFEFF00@, & F0-FF
```

```
    (20);
    ADDRESS;
    DISPLAY "Enter lower-case text to be translated to upper
    case";
    ACCEPT DATA_IN;
    TRANSLATE DATA_IN WITH TRANSLATE_TBL TO DATA_OUT;
    DISPLAY DATA_OUT;
END;
```

UNLOCK

The function of UNLOCK is to unlock a locked block of a shared disk or diskpack file. Figure 5-68 describes the format of the UNLOCK statement.

```
UNLOCK file-name ;
```

Figure 5-68. Format of the UNLOCK Statement

Under MCP releases prior to MCP/VS 2.0, the use of LOCK is valid only when the MCP SHRD option is set. In any case, the FILE declaration must include the SHARED attribute.

WAIT

The WAIT statement suspends the execution of the object program either for a specified period of time, or until one or more conditions are true. Figure 5-69 shows the format of the WAIT statement.

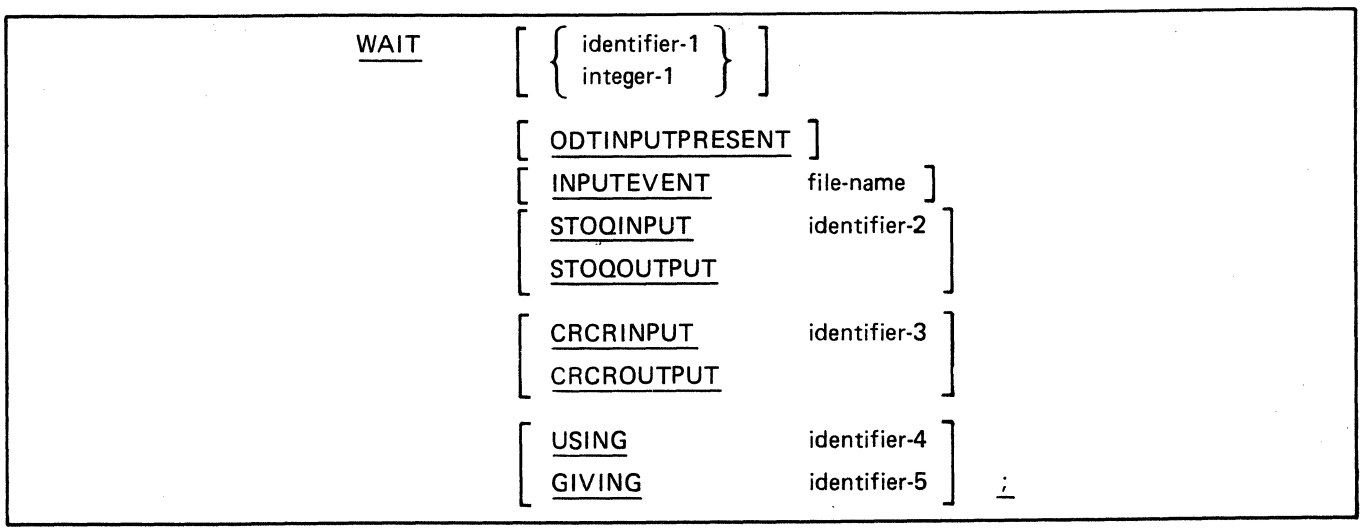


Figure 5-69. Format of the WAIT Statement

A WAIT with no options specified suspends the program for 86400 seconds (24 hours).

Not more than one identifier-1 or integer-1 can be specified, and if declared, must be the first item in the list. Identifier-1 must be an INTEGER of five digits or less. The maximum value for identifier-1 and integer-1 is 86399.

When ODTINPUTPRESENT is specified, execution of the object program is suspended until an AX message is received for the program.

When INPUTEVENT is specified, the program is suspended until the event becomes TRUE. File-name must be the name of a file declared in a FILE statement.

When STOQINPUT is specified, the program is suspended until a STOQUE entry is available for retrieval. When STOQOUTPUT is specified, the program is suspended until storage space is available. Identifier-2 specifies the STOQUE Parameter Block for the STOQUE queue to be checked.

When CRCRINPUT is specified, the program is suspended until the sender is ready to transmit the data; when CRCROUTPUT is specified, the program is suspended until the receiver is ready to receive. Identifier-3 specifies the name of the program which is the object of the FILL statement, declared as ALPHA with a length of six.

If any event in the list is TRUE, the WAIT is terminated and control is passed to the next executable statement.

If no event is TRUE during initial scan of the events, program execution is suspended until any event listed becomes TRUE, at which time the WAIT statement is terminated and control is passed to the next exectuable statement.

When the USING clause is not specified, each event in the list, from first to last, is tested for a TRUE condition. This process is repeated until an event becomes TRUE. If the USING clause is specified, the value referenced by identifier-4 determines the begin point for the TRUE condition search, one-relative from the beginning of the list. If identifier-4 is zero, evaluation begins with the first entry in the list. In any event, all entries are evaluated until a TRUE condition is found. Identifier-4 must be an INTEGER of two digits.

When GIVING is specified, the item referenced by identifier-5 is set to the position of the event that terminated the WAIT statement. Identifer-5 must be an INTEGER of two digits.

Examples:

```
WAIT    50;

WAIT    STOQINPUT    STOQ_PARAMS;

WAIT    MAX_TIME

    ODTINPUTPRESENT
    CRCRINPUT       PNAME
    CRCROUTPUT      NAME2
    USING           POINTER
    GIVING          HIT;
```

WHILE

WHILE specifies a condition under which a DO statement may be executed: the DO is executed while the condition is true. The WHILE condition is evaluated before the DO is executed.

Refer to DO for syntax and a detailed explanation.

WRITE

The function of WRITE is to release a logical record for an output file. It is also used to position forms vertically in the printer. For mass storage files, the WRITE statement also allows branching to a routine if the contents of the associated KEY are found to be invalid.

Figure 5-70 describes the format of the WRITE statement.

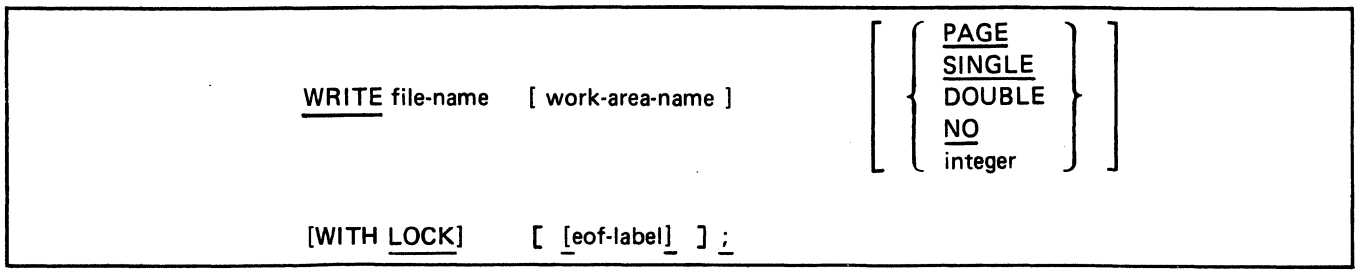


Figure 5-70. Format of the WRITE Statement

An OPEN statement for a file must be executed prior to executing the first WRITE statement for that file.

If the work-area-name is specified, the corresponding FILE declaration must have requested a WORKAREA. The WORKAREA indicated by the FILE declaration will be used until explicitly changed by the execution of a WRITE statement containing a different WORKAREA name.

The options for vertical positioning of the printer are:

- When PAGE is used, a skip to a new page is forced, that is, a skip to CHANNEL 1.
- SINGLE and DOUBLE allow for single-spacing or double-spacing. DOUBLE is the default if no option is specified.
- Overprinting is accomplished by use of NO.
- Integer-1 must be a 2 digit integer with a positive value between 01 and 11. It specifies the channel number to which the printer will be advanced.

A WRITE to a shared disk or diskpack file must be preceded by a LOCK. Since LOCK destroys the record area, the record must be initialized after the LOCK.

The WITH LOCK option applies only to shared disk and diskpack processing, and cannot be specified unless the current record has previously been locked by the program. When this option is specified, the current record is written, but the block is not UNLOCKed; therefore, the block is not available to other programs. If the WITH LOCK option is not used in shared disk processing, a record is written and if the block was locked, it is unlocked. Under MCP releases prior to MCP/VS 2.0, these constructs may be used only when the MCP SHRD option is set. In any event, the FILE declaration must include the SHARED attribute in order to use these constructs.

The record can be released by either a WRITE without the LOCK option, or by use of the UNLOCK statement.

A simple WRITE to a shared file results in a write followed by an unlock operation.

For disk and diskpack files which are being accessed in a SERIAL manner, the eof-label branch is executed when the end of the last segment of the file (last record) has been reached and another attempt is made to WRITE to the file. For RANDOM files, the eof-label branch will be executed whenever the value of the KEY field is outside the defined bounds of the file.

If the size and blocking of records being accessed in a RANDOM manner is such that a WRITE statement must place a record into a block without disturbing the other contents of the block, then an implicit SEEK will be given to load the block desired (if an explicit SEEK has not been given). If the file is being processed IO (INPUT/OUTPUT), either an explicit or implicit SEEK for a READ statement will suffice to load the block.

If the value of the KEY field is changed after a SEEK and prior to the WRITE statement, an implied SEEK will be performed and the WRITE will use the new record address as the output record area.

The shortest allowable records which can be written on 7 and 9 channel magnetic tape units are 8 and 18 bytes respectively.

When the WRITE statement is executed at object time, the logical record is released for output and is no longer available for referencing by the object program. When the blocking area becomes full, or partially full at EOJ or EOF, the object program will transfer control to the MCP to cause the block to be physicallywritten. Short blocks or records which are written during EOJ or EOF will be of no programmatic concern to the user when the file is processed as input at a later time.

ZIP

The function of ZIP is to cause the MCP to execute a control instruction contained within the operating object program.

Figure 5-71 describes the format of the ZIP statement.

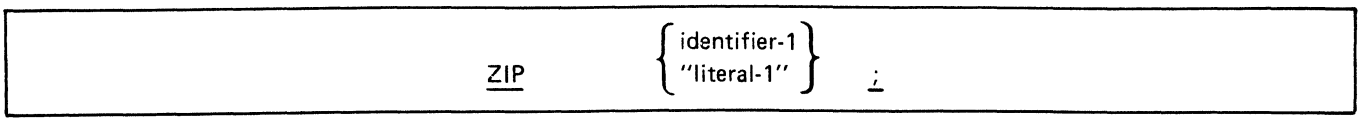


Figure 5-71. Format of the ZIP Statement

Identifier-1 must be assigned a value equivalent to the inforamtion contained in a MCP Control instruction.

Literal-1 allows coding of the MCP control instructions within the ZIP format. The quote marks are required.

If the control text is longer than 72 characters, the text must start with a period. The control text must be terminated by a period.

SECTION 6
COMPILER DIRECTING STATEMENTS

GENERAL

The following BPL constructs are for use as compiler directing statements. For ease of reference, they are listed alphabetically in the following pages.

Conditional Compiling

The function of Conditional Compiling is to allow the user to either include or exclude certain source statements depending on a flag setting.

Figure 6-1 describes the format of the conditional compile double dollar-sign record.

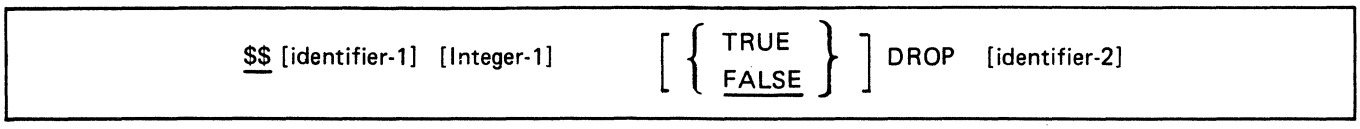


Figure 6-1. Format of the Conditional Compile, Double Dollar-Sign Record

The BPL compiler makes available to the user, 50 flags to be set or reset as needed (flags are number 01 through 50).

These flags are set or reset by a $ SET or $ RESET record that includes the $ key word FLAG followed by one or more flag numbers not separated by commas and delimited only by another $ option or the end of the record.

The double dollarsign ($$) must be in columns 1 & 2 of the record.

Identifier-1 is a one character identifier and must be in column 3 if used.

Identifier-1 is used to overlap the conditional compile, see example below.

Integer-1 is the FLAG number as specified in the regular dollar sign ($) record. Integer-1 must not start in column 3 if used. This FLAG (integer-1) will be tested for a TRUE or FALSE condition. If TRUE, or neither, the test will be for ON status of the FLAG. If FALSE the test will be for OFF status.

The word DROP is for documentation only. Identifier-2 has the same restrictions and is used in conjunction with identifier-1.

If the $$ record is blank or contains only $$ identifier-1, all records unconditionally enter the compiler.

All $$ records and symbolic records are placed on the new symbolic output file.

The output listing will reflect *all* $$ records, and *all* symbolic records. ANY records not compiled will be so noted as will any records changed or conditionally compiled. This will highlight conditional coding whether patched or not. To suppress listing of uncompiled records, set the $DLIS compiler control option.

Example:

```
$ SET FLAG 02 48
$$ 02 TRUE DROP B                    & with 02 TRUE all records
                                     & to $$B will be ignored
    .
    .
    .
$$48 TRUE DROP                       & if 02 FALSE 48 TRUE , record
    .                                & to $$ would be dropped
    .
    .
$$                                   & stops 48 condition
    .
    .
    .
$$B                                  & stops 02 condition
```

For low-volume conditional compiling, see also the IFF statement, elsewhere in the section.

@LIBR

The function of LIBR is to create library files for subsequent use by other programs. Figure 6-2 shows the format of the @LIBR statement.

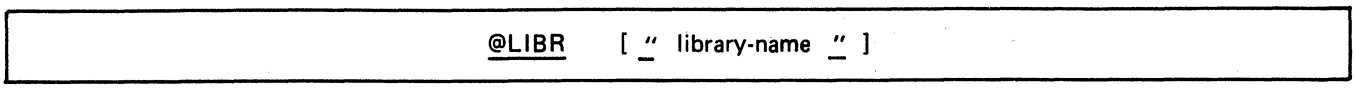


Figure 6-2. Format of the @LIBR Statement

@LIBR must be in columns 1-5 of the record.

Library-name must be enclosed in quotes and must be a non-numeric literal no more than 6 characters long.

Library-name if present will cause a new library file to be created from this point. If not present the previous library file will be closed at this point.

All source statements including compiler control records will be placed in a disk file with the file-identifier as specified by library-name.

Examples:

```
@LIBR "LIB1"                    & Create LIB1 from this point
  statement-1
      .
      .
      .
  statement-N
@LIBR "LIB2"                    & Close LIB1 create LIB2
  statement-1                   & from this point
      .
      .
      .
  statement-N
@LIBR                           & Close LIB2.
```

Library names may not start with a blank or hyphen (-). The source data used to create an original library file will also be compiled into the object program at the point of appearance.

All assigned library-names must be unique to other library-names contained in the library to preserve the integrity of the BPL library system.

Library files to be used in BPL programs can be created by a user program which creates a card image file blocked five or nine on disk. Library files created with @LIBR are closed with CRUNCH and blocked nine.

@PAGE

The function of PAGE is to cause the compiler to advance to the top of the next page of the compile listing. Figure 6-3 shows the format of the @PAGE statement.

```
@PAGE
```

Figure 6-3. Format of the @PAGE Statement

@PAGE must be in columns 1-5 of the line. No other BPL statement should appear on the same line.

@ICM Declaration

The function of the ICM declaration is to create Type I Independently Compiled Modules (ICM) for subsequent compilation by other programs. Figure 6-4 shows the format of the ICM declaration.

```
@ICM   [ " subroutine-name " ]
```

Figure 6-4. Format of the ICM Declaration

@ICM must be in columns 1-4 of the line. No other BPl statements should appear on the line.

Subroutine-name must be enclosed in quotes and must be no more than 6 characters long.

Subroutine-name, if present, will cause a new Type I ICM to be created from this point. If not present, the Type I ICM currently being built will be concluded, and normal program compilation will continue.

Example:

```
? COMPILE ADDIT BPL LIBRARY
? DATA CARD
? CARD LIST
BEGIN
@ICM "ADDONE"
PROCEDURE ADDONE (INFIELD, OUTFIELD);
INTEGER INFIELD (6),
        OUTFIELD (6);
BEGIN
OUTFILED := INFIELD + 1 ;
END;
@ICM
END;
? END
```

NOTE

Type I ICMs are for use in FORTIV and XFORTN FORTRAN programs only.

See APPENDIX D (Independently Compiled Modules) for further information.

IFF Conditional Compiling

The function of IFF conditional compiling is to allow an easy way to determine which of two groups of source statements in a group of symbolics is compiled. Figure 6-5 shows the format of the IFF statement.

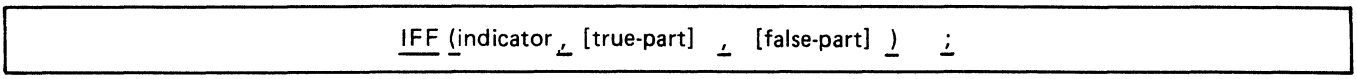


Figure 6-5. Format of the IFF Statement

The indicator must be present and should be a DEFINE indentifier. If the defined value of the indicator is false (%F0%,%40%, %00%), the false-part symbolics are compiled, otherwise the true-part symbolics are compiled. If the indicator is not found to be defined at the time the IFF is encountered, the true-part symbolics will be compiled.

The true-part or false-part need not be present if the desire is presence or absence of symbolics, however, the separating commas must always be present.

If the true-part or false-part contain commas, the entire true-part or false-part should be contained within parentheses. The true-part or false-part may not contain unmatched parentheses.

For readability, IFF should be used for low volume conditional compiling. Larger amounts of symbolics should use the $$ conditional compiling feature.

Example:

```
BEGIN
DEFINE KB30 = 1#;                                  & MEANS TAKE TRUE-PART
CONTROL MEMORY :=IFF (KB30,58000,88000);           & COMPILES CONTROL
                                                   & MEMORY :=58000;
IFF (KB30,(INTEGER A (6),B (6)),                   & COMPILES INTEGER A(6),
                                                   & B(6);
        (INTEGER A(10), B(10)));                   & COMMAS MUST BE WITHIN
                                                   & PARENS
IX1 :7;
IX2 := IFF (TRUEPART,A.3, A.7);                    & COMPILES IX2 :=A.3
                                                   & BECAUSE TRUEPART IS
                                                   & RETURNED AS THE
                                                   & SYMBOL "TRUEPART" AND
                                                   & IS NOT FALSE
END;
```

NOTE

The changing of KB30 to a define of 0, would create a program 88000 digits in size with 2 variables, each 10 digits in length. The assignment of IX2 would not be affected.

SECTION 7
DATA COMMUNICATIONS

GENERAL

This section deals with the BPL constructs required to activate the data communications equipment as defined by the FILE declaration.

The specific formats together with a detailed discussion of the restrictions and limitations associated with each, appear on the following pages in alphabetic sequence.

NOTE
The Data Communications module of the operating system must be present to use any of the statements in this section.

ACCEPT

The function of ACCEPT is to permit the entry of low-volume data from a remote SPO.

NOTE
Remote SPOs are not supported by MCP/VS operating systems with a release level of 2.0 or later. Programs that use DATACOMM ACCEPT statements do not work correctly when executed under an MCP/VS operating system with a release level of 2.0 or later. Effective with the 2.0 release of BPL, DATACOMM ACCEPT is deleted from the BPL language, and its use will result in a syntax error.

Figure 7-1 shows the format for this statement.

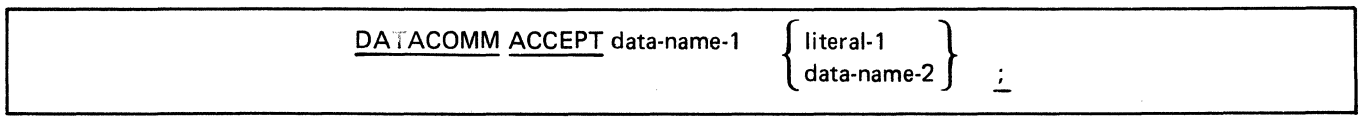


Figure 7-1. Format of DATACOMM ACCEPT

This statement causes the operating object program to halt and wait for appropriate data to be entered through a remote SPO. The remote SPO operator responds to an ACCEPT halt by keying in the following message:

? <mix-index> AXdata-required

If a blank appears between the AX and data-required, the blank character will be included in the data-stream.

When the object program executes an ACCEPT statement, the information will be transmitted from the remote SPO keyboard into memory locations assigned to data-name-1. The data-name must define a UA field.

Because of the inefficiency of entering data through the remote keyboard, this technique of data transmission should be used sparingly and solely restricted to low-volume data.

The values of literal-1 or data-name-2 describe the name of the remote SPO from which data will be transmitted into the receiving-data-name and must be the alpha mnemonic name assigned to that specific remote SPO (this is the adapter ID which is specified in the MCP's UNIT record of the System Specification Deck). Literal-1 and data-name-2 cannot exceed six characters in length; if less than six characters, it must be followed by a blank.

Literal-1 can also represent the channel and unit specification of the remote SPO in the format cc/u where cc represents the channel and u represents unit number. A trailing blank must follow unit number.

If the named device is not a remote, or not a remote SPO, or not logged-in, the ACCEPT will be directed to the local SPO.

CANCEL

The function of CANCEL is to conditionally/unconditionally cancel the prior IO descriptor. It stops the flow of data to or from a remote device with or without disconnecting a dial line and with or without signalling a break to the operator of the remote device.

Figure 7-2 shows the format of this statement.

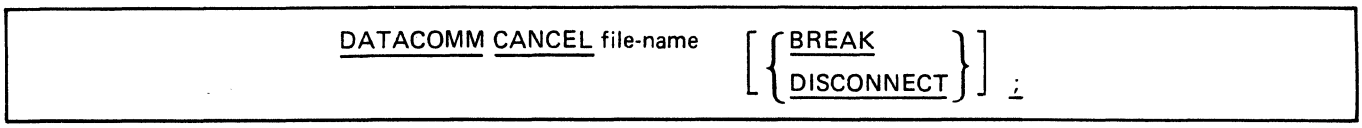


Figure 7-2. Format of DATACOMM CANCEL

The CANCEL file-name (no option) unconditionally cancels the prior descriptor verb and renders the Data Communications File insensitive to data transmission both to and from the remote device.

When the BREAK option is used, the prior descriptor verb is unconditionally cancelled and the Data Communications File is rendered insensitive to data transmission to and from the remote device as well as transmitting a break to the remote device. However, the BREAK condition is only applicable to terminals and data sets that are designed to respond to a BREAK condition, (i.e., B 9350).

When the DISCONNECT option is used, the prior descriptor is unconditionally cancelled. The Data Communications file is insensitive to data transmission (or requests), and the telephone line is disconnected. This applies only to switched line networks.

CONDCANCEL

CONDCANCEL requests that a conditional cancel of the current I/O operation be performed on the designated file.

```
DATACOMM CONDCANCEL identifier-1 ;
```

Figure 7-3. Format of DATACOMM CONDCANCEL

Identifier-1 is the name of a previously defined file.

The cancel is ignored if no I/O is in progress, if the adapter is sending data to the device, or if data from the device is being received.

The operation POLL is cancelled only if the device currently being polled does not return a positive response.

DISPLAY

The function of DISPLAY is to provide for the printing of low-volume data, error messages, and operator instructions on a remote SPO.

NOTE
Remote SPOs are not supported by MCP/VS operating systems with a release level of 2.0 or later. Programs that use DATACOMM DISPLAY statements do not work correctly when executed under an MCP/VS operating system with a release level of 2.0 or later. Effective with the 2.0 release of BPL, DATACOMM DISPLAY is deleted from the BPL language, and its use will result in a syntax error.

Figure 7-4 describes the format of this statement.

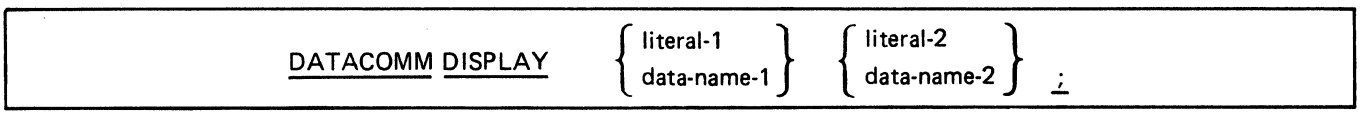


Figure 7-4. Format of DATACOMM DISPLAY

Data-name-1 and literal-1 (and their associated series) are specified as being the area within an object program from which data is to be transmitted to a receiving remote SPO.

The DISPLAY statement causes the contents of each operand to be transmitted from the MCP SPO queue to ensure that an operational program is not delayed while the remote SPO message is being printed.

Data-name-1 may be subscripted and can be represented as INTEGER or ALPHA.

Up to 60 characters may be contained in a literal, or data-name to be DISPLAYed.

The value contained in literal-2 or data-name-2, describes the name of the remote SPO to which data will be transmitted and *must* be the alpha mnemonic name assigned that specific remote SPO as defined in the MCPs UNIT record of the Systems Specification Deck (the adapter identifier).

Literal-2 can also represent the channel and unit specification of the remote SPO in the format cc/u; where cc represents the channel and u represents the unit number. A trailing blank must follow the unit number.

ENABLE

The function of this verb is to disconnect the telephone line for dial lines, recognize a ringing signal, or recognize an enquiry (ENQ) from an appropriate remote device.

Figure 7-5 describes the format of ENABLE.

```
DATACOMM ENABLE [EXTENDED data-name-1] file name [ [action-label] ] ;
```

Figure 7-5. Format of ENABLE

File-name must have been OPENed before an ENABLE can be executed. Once the file-name has been ENABLEd a WAIT statement must be used. Reference the WAIT verb in this section.

The EXTENDED option causes the INTERROGATE and INTERROGATE ADDRESS operations to be performed at the end of the ENABLE operation.

The ENABLE allows device recognition by the system by either depressing the Inquiry Key (ENQ) if the device is connected on leased lines or direct connect, or by dialing the system's telephone number of the device is on dialed lines.

The ENABLE statement will cause dial telephone lines to be disconnected and will recognize input requests from the device in the form of a telephone ringing signal.

The ENABLE statement for leased or direct connect lines will recognize input requests from the remote device in the form of an inquiry (ENQ).

Data-name-1 is used only with the EXTENDED option. It is the response area for the INTERROGATE and INTERRROGATE ADDRESS operations. Data-name-1 must be greater than or equal to 26 digits and will be in the following format:

I/O Character Count	-	6 UN
Result Descriptor	-	4 UN
Expanded R/D	-	16 UN

For a description of the action-label, see the WAIT statement.

FILL

The function of this verb is to initiate a specified type of I/O and allow a program to run without waiting for an I/O to be completed. It is useful when a program does not require the input data to continue processing.

Figure 7-6 shows the format if FILL.

```
DATACOMM FILL [EXTENDED DATA-name-1] file-name [NOTIMEOUT] [PRESETSTX]
              [DIAL] [DELETEETX]

                                                  { WRITE          }
                                                  { READ           }
[POLL] [VOICE] [TONE] [STREAM] [FLIPFLAG]         { WRITERREAD     }
                                                  { WRITETRANSREAD }
                                                  { WRITEREADTRANS }

[ [action-label] ] ;
```

Figure 7-6. Format of FILL

FILL causes the operation to be initiated. The program must accomplish a READ to move the required data to the work area when using STREAM mode.

The EXTENDED option causes the INTERROGATE and INTERROGATE ADDRESS operations to be performed at the end of the FILL operation.

Data-name-1, see ENABLE statement.

If the NOTIMEOUT option is used, the time-out feature is inhibited on a READ construct or the READ portion of a WRITEREAD, WRITETRANSREAD or WRITEREADTRANS.

If the option PRESETSTX is used, the first character transmitted or received is considered text and is used in generating the longitudinal redundancy check character. The PRESETSTX function will be automatically pre-set by the first character.

If the option DIAL is used, a dial number is accessed from memory starting at the record-name of the record-description for the file-name specified. The dial numbers must be declared as INTEGER and the dial number field must be terminated by the binary control code of 1100 (undigit 12) which can be represented in BPL as an undigit literal @C@. The total number of digits comprising the dial number must be even, and a filler digit with a zero value must be inserted after the undigit literal if necessary. The rest of the record-description for file-name will describe the input data from the control code or filler, whichever is applicable, to the end of the record description.

If the option DELETEETX, WRITE or WRITEREAD is used, the control code EOT is transmitted but the control code denoting end-of-text (ETX) is not transmitted. This sequence causes the WRITE command of the WRITE portion of a WRITEREAD command to become complete. The ETX function is ignored and the longitudianl redundancy check (LRC) character is not generated or sent. This option may be used for polling operations.

If the VOICE option is used, the voice response adapters are ENABLEd automatically. Characters are passed from memory to the voice responder and are used as voice track addresses. The resulting signals from the voice responder are sent over the line.

If the TONE option is used, the tone leads on the Tone Data Sets are activated. Characters received from memory by the I/O adapter are sent to the tone leads as follows:

- "B" characters to the 2025 hz lead.
- "A" characters to the 1017 hz lead.
- All others to the silent lead.

The tones or silence will continue for 300 milliseconds per character sent.

The POLL option is only used with WRITEREAD. The output area will consist of a series of contiguous polling sequences. The WRITEREAD will be executed with the first such polling sequences to be transmitted to the remote device. If a message, other than a negative response (EOT) is returned, the entire operation terminates when the control character denoting end of text (ETX) is sensed at the end of the input message. If an exception condition (such as time-out) exists, the I/O will also complete. A negative response to a poll (EOT) does not enter main memory, and another polling sequence is automatically initiated (WRITEREAD). If only negative responses are returned (EOT to every poll), the operation will also terminate when the ending control code ETX is sensed, telling the hardware that the polling sequences have been exhausted. If a message is sent from the remote, it will be written in the area following the polling sequence which received the response, thus overlaying the remaining successive sequences.

If the STREAM option is specified, the information is transferred into/from ascending memory locations starting at the record-name of the record description for the specified file-name. See the STREAM option of the READ statement in this section for further information.

The FLIPFLAG option initiates a write/read check.

FILL statements normally are initiated to several Data Communications files and are followed by a WAIT statement. See WAIT statement discussion in this section.

If the action label option is not specified and the I/O goes complete, control transfers to the instruction following the WAIT statement.

STREAM with POLL indicates recirculating poll.

INTERROGATE

The function of INTERROGATE is to obtain a result descriptor representing the operational status of a remote device.

Figure 7-7 shows the format of INTERROGATE.

```
DATACOMM INTERROGATE [ADDRESS] file-name data-name-1 ;
```

Figure 7-7. Format of DATACOMM INTERROGATE

Data-name-1, when ADDRESS is not specified, must be defined as 16 UN (INTEGER (16)).

The result descriptor contains 16 digits which indicate conditions that occurred during an I/O operation. The digits of information within the result descriptor are assigned the following meanings when turned "ON".

Table 7-1. Result Descriptor Digits

Digit Number	Meaning
1	Operation complete.
2	Exception condition.
3	Data - set not ready.
4	Data error.
5	Abandon call retry (ACR).
6	Cancel complete.
7	End-of-transmission (EOT).
8	Attempt to exceed maximum address.
9	Time out.
10	Memory parity error.
11	Write error.
12	Carrier loss.
13	Stream complete.
14 through 16	Reserved.
4 and 5	Data loss.
6 and 7	Break detected.

ON status is indicated by a value of 1. OFF status is indicated by a value of zero. Digits are independent of one another and can reflect varied combinations.

Table 7-2 explains the status of the result descriptor digits shown in table 7-1.

Table 7-2. The Status of Result Descriptor Digits

Digit Number	Status
1	Always ON if the attempted operation was completed.
2	Will be ON if any combination of 3 through 16 are ON. This is the test position to see if any exception exists. If this position is ON by itself, a partial complete condition exists due to the use of READ STREAM and will not occur in any other situation.
3	Will be ON if the single line control or the local Data Set is not ready and the operation will be terminated. For multiline control the digit is set ON in the channel result descriptor unless it occurs during an operation, in which case it is set ON in the adapter result descriptor.
4	If a data error (message or character parity) occurs, a READ operation continues until terminated in a normal manner. The phone line is not disconnected. Attempts to exceed maximum address, time out End-of-Text (ETX), or End-of-Transmission (EOT) can also occur.
4 and 5	If data loss (missed memory access or MLC cycle), a READ operation continues until terminated in a normal manner. Attempts to exceed maximum address, time out, End-of-Text (ETX), or End-of-Transmission (EOT) can also occur. The phone line is not disconnected. A WRITE operation is terminated immediately and digit 11 is set.
5	If an abandon call retry condition exists, this position will be set ON and the telephone line is disconnected. This position is also set on by a timer in the Automatic Calling Unit.
6	If a cancel complete condition exists, this position will be set ON and CANCEL is initiated.
6 and 7	If a break is detected, these positions will reset ON for a WRITE operation only and the operation is immediately terminated. The telephone line is not disconnected.
7	If the End-of-Transmission exists, this position is set ON and the telephone line is disconnected.

Table 7-2. The Status of Result Descriptor Digits (Continued)

Digit Number	Status
8	If an attempt to exceed maximum address exists, a READ operation will initiate a time out and wait for a control code denoting End-of-Text (ETX). This position along with position 7 will be set ON if an (EOT) is received before time out. This position and position 9 will be set ON if time out occurs without ETX or EOT. A WRITE operation is immediately terminated and this position along with position 11 is set ON. The telephone line is disconnected in each case.
9	If time out exists, this position is set ON and the telephone line is not disconnected. Time out occurs on DATACOMM READ and DATACOMM WRITE instructions.
10 and 11	If a memory parity error exists, these positions are set ON and the telephone line is not disconnected. Memory parity error occurs only during a WRITE operation, and will immediately terminate the operation.
12	A READ operation is terminated in a normal manner. The phone line is not disconnected.
13	Used in multi-line control to check end-of- stream, when operating in STREAM mode.
14,15 and 16	Reserved.

If the ADDRESS option is specified, data-name will contain the number of characters transmitted to and/or from the current buffer of the file. Counting begins when the descriptor isinitiated and continues until it is complete. Data-name must be defined as INTEGER (6). If the I/O is not complete on the current buffer, the INTERROGATE ADDRESS is ignored.

READ

Requests "Read to control" operation on designated file. The program is suspended until the operation is completed.

Figure 7-8 describes the format of READ.

```
DATACOMM READ [EXTENDED data-name-1] file-name [NOTIMEOUT]

[PRESETSTX] [STREAM] [DIAL]        [ [eof-label] ]    ;
```

Figure 7-8. Format of READ

Loading will continue until an ending code such as End-of-Transmission (EOT), End-of-Text (ETX) or End-of-Transmission Block (ETB) is detected, or until the buffer is filled.

The EXTENDED option causes the INTERROGATE and INTERROGATE ADDRESS operations to be performed.

Data-name-1, see ENABLE statement.

NOTIMEOUT, refer to the FILL statement.

PRESETSTX, refer to the FILL statement.

If the STREAM option is used, the information is written into the record-name of the record-description for the file-name specified. The record-description entry must define at least 200 digits (100 bytes). A control code denoting End-of-Text (ETX) will terminate the operation. The use of a filler entry after the End-of-Text (ETX) control code will not be the last position in that entry.

DIAL, see the FILL statement.

eof-label, see WRITEREAD statement.

READY

Specifies that the STREAM mode buffer is empty and ready to READ more date or that the buffer is full and ready to WRITE.

Figure 7-9 shows the format of READY.

```
DATECOMM READY BUFFER file-name ;
```

Figure 7-9. Format of READY

Ready is executed implicitly by the MCP as READ and WRITE requests are made to file-name. READY must be received by the I/O control within the required period or data can be lost.

TRANSTBL

Requests the transfer of data communications translate tables from the MCP to the requester.

Figure 7-10 shows the format of TRANSTBL.

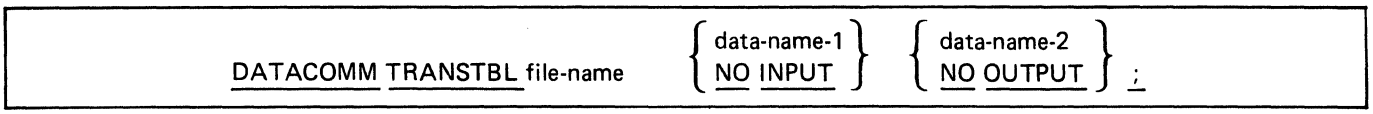


Figure 7-10. Format of TRANSTBL

Data-name-1 is the input table area and data-name-2 is the output table area.

Both must be at a mod 4 address, but not necessarily at mod 1000.

The reserved word NO followed by either INPUT or OUTPUT will allow either the input or the ouput table area to be zero when not needed. Please note that BOTH cannot be zero.

The NO OUTPUT option must be used if non-standard translation is specified.

WAIT

The function of WAIT is to suspend an object until a response to an ENABLE or the completion of a FILL statement occurs, and (or) to suspend an object program for a specified number of seconds.

Figure 7-11 shows the format of WAIT.

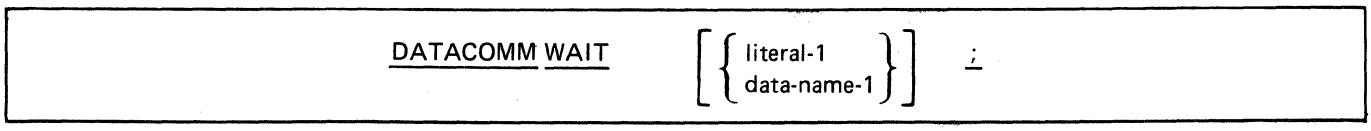


Figure 7-11. Format of WAIT

Literal reflects the number of seconds that the object program is to be suspended. The maximun WAITing period is 23 hours, 59 seconds, and 59 seconds (86399).

If the literal or data-name option is not used, the WAIT will suspend the object program until a response to an ENABLE or the completion of a FILL request occurs. At that time control is returned to the action label in the object program given with the ENABLE or FILL request for that device. If no action label was given, the processing contines at the statement immediately following the WAIT request.

The WAIT request is programmed to place the object program in WAIT status until a specified number, data-name-1 or literal, of seconds have elapsed or until a previously initiated FILL or ENABLE operation becomes complete, whichever occurs first. Data-name-1 must be described as INTEGER (5).

The following rules apply when WAIT is used in conjunction with the ENABLE or FILL constructs:

- If an ENABLE and/or FILL statement contains as action label and either statement comes "true", the object program will be reinstated at the appropriate action label.
- If the action label option is omitted and an ENABLE and/or FILL statement comes "true", the object program will be reinstated at the next instruction following the WAIT statement.
- If the action label is omitted, and the literal or data-name option is used, the object program must determine how it got to the next instruction. That is: did a FILL ENABLE come "true" or did the WAIT time expire? An INTERROGATE of all ENABLEd or FILLed files will have to be performed to determine the answer.
- If more than one I/O complete occurs at a given time, the ENABLE and FILL completes are serviced in the order of priorites established at the time of the COLD START loading.

WRITE

You can use WRITE to request a "Write to Control" operation to a designated file.

Figure 7-12 shows the format of WRITE.

```
DATACOMM WRITE [EXTENDED data-name-1] file-name [DELETEETX]

[PRESETSTX] [STREAM] [DIAL] [VOICE] [TONE] [ [eof-label] ] ;
```

Figure 7-12. Format of WRITE

Data will be passed until a control code denoting End-of-Text is detected in the file-name. Control code characters denoting End-of-Text vary depending on the type of line adapter and remote being used.

The EXTENDED option causes the INTERROGATE and INTERROGATE ADDRESS operations to be performed at the end of the WRITE operation.

Data-name-1, see ENABLE statement.

DELETEETX, see the FILL statement.

PRESETSTX, see the FILL statement.

STREAM, see the READ statement.

DIAL, see the FILL statement.

VOICE, see the FILL statement.

TONE, see the FILL statement.

eof-label, see the WRITEREAD statement.

WRITEREAD

The function of WRITEREAD is to pass data to a remote device from memory locations and, when successfully completed, to cause data to be read from the remote device and passed to appropriate memory locations.

Figure 7-13 shows the format of WRITEREAD.

```
DATACOMM WRITEREAD [EXTENDED data-name-1] file-name [NOTIMEOUT]

[DELETEETX] [PRESETSTX] [VOICE] [DIAL] [TONE] [POLL] [STREAM]

[ [ eof-label ] ] ;
```

Figure 7-13. Format of WRITEREAD

Data will be passed to the remote device from ascending memory locations starting at file-name-1 and will continue wuntil a control code denoting End-of-Transmission (EOT) or End-of-Text (ETX) is detected. A READ will then be initiated on the remote device and the data will be passed to ascending memory locations immediately following the End-of-Transmission (EOT) or End-of-Text (ETX) control code wihich terminated the WRITE and will continue until an End-of-Text (ETX) control code from the remote device is encountered. Each portion of the message being written and read must be terminated by an End-of-Transmission (EOT) or End-of-Text (ETX) control code.

The EXTENDED option causes the INTERROGATE and INTERROGATE ADDRESS operations to be performed at the end of the WRITEREAD operation.

Date-name-1, see ENABLE statement.

NOTIMEOUT, see the READ statement.

DELETEETX, see the WRITE statement.

PRESETSTX, see the READ statement.

VOICE, see the WRITE statement.

DIAL, see the WRITE statement.

TONE, see the WRITE statement.

POLL, see the WRITE statement.

STREAM, see the WRITE statement.

If the eof-label option is used, the End-of-Transmission control code received will cause the program to accomplish the indicated actions. Control will pass to the next instruction in the absence of an eof-label.

Refer to the FILL and WAIT constructs, this section, for requirements of FILL with WRITE and FILL with WRITE and WAIT.

STREAM with POLL indicates recirculating poll.

WRITEREADTRANS

The function of WRITEREADTRANSparent is to pass data to a remote device (normally a computer) from memory locations and, when successfully completed, to cause data to be read from the remote device and passed to appropriate memory locations, and terminating at the end of the record-description without passing an End-of-Transmission (EOT) or End-of-Text (ETX) control code.

Figure 7-14 shows the format of WRITEREADTRANS.

```
DATACOMM WRITEREADTRANS [EXTENDED data-name-1] file-name [NOTIMEOUT]

[DIAL]        [ [ eof-label ] ] ;
```

Figure 7-14. Format of WRITEREADTRANS

Data will be passed to the remote device from ascending memory locations starting at file-name record and will continue until a control code denoting End-of-Transmission (EOT) or End-of-Text (ETX) is detected. A READ will then be initiated on the remote device and the data will be passed to ascending memory locations beginning with the location immediately following the End-of-Transmission (EOT) or End-of-Text (ETX) control code which terminated the WRITE and will continue until the end of file-name record. The attempt to exceed Maximum Address in the Result Descriptor will not be turned ON when the end of file-name record is reached.

The EXTENDED option causes the INTERROGATE and INTERROGATE ADDRESS operations to be performed at the end of the WRITEREADTRANS operation.

Data-name-1, see ENABLE statement.

NOTIMEOUT, see the FILL statement.

DIAL, see the FILL statement.

eof-label, see the WRITEREAD statement.

This statement is normally used for communication with remote computers.

WRITETRANSREAD

The function of WRITETRANSREAD is to pass data to the remote device (normally a computer) until the end of the record description is reached and, when successfully completed, to cause data to be passed from the remote device to memory locations, starting at the end of the record-description and continuing until an End-of-Transmission (EOT) or End-of-Text (ETX) control code is detected.

Figure 7-15 shows the format of WRITETRANSREAD.

```
DATACOMM WRITETRANSREAD [EXTENDED data-name-1] file-name [NOTIMEOUT]

[DIAL]          [ [eof-label] ]  ;
```

Figure 7-15. Format of WRITETRANSREAD

The READ portion of this statement will continue passing data until an End-of-Transmission (EOT) or End-of-Text (ETX) control code is detected, or will terminate the flow when the location, ending file-name record address + 199 is reached.

The EXTENDED option causes the INTERROGATE and INTERROGATE ADDRESS operations to be performed at the end of the WRITETRANSREAD operation.

Data-name-1, see ENABLE statement.

NOTIMEOUT, see the FILL statement.

DIAL, see the FILL statement.

eof-label, see the WRITEREAD statement.

This statement is normally used for communication with remote computers.

The end of the record-area may be programmatically altered by use of the KEY data-name must be defined as INTEGER (6). The value contained there-in will be used when the I/O is initiated to determine the number of characters in the record area. The ending address of the result descriptor will be adjusted accordingly. (Refer to INTERROGATE, elsewhere in this section, for information on the result descriptor.)

SECTION 8
PORT FILES

GENERAL

Port files permit programs running on the same or different processors to exchange data. The port file is a means of interprogram communication. Ports are somewhat like the older Core-to-Core and Storage Queue mechanisms but differ from those in that a program can communicate through a port file with another program on any Unisys V Series, B 2000/B 3000/B 4000, A Series, B 5000/B 6000/B 7000, B 1000, or B 900 CMS systems.

Port files are designated as local-only ports when both programs are executing on the same processor, and as remote ports when the programs are on different processors. The only difference between the two is that to use remote ports, BNA software must be on both systems, while to use local-only ports, only the current operating system and the STOQ option are required on V Series or B 2000/B 3000/B 4000 Series systems.

A port file contains from one to several hundred subports. Subports are themselves the individual files through which dialogues occur. Each subport shares the attributes of its parent port, and has its own attributes as well.

To use port files, each program declares a port file and certain subport attributes to be used as matching criteria. If remote ports are to be used, each program specifies the hostname of the remote port file with which it wishes to communicate. If, when the subport is OPENed, the corresponding file is found, a dialogue is established between the two programs. Records written to the subport of either program can be read from the subport of the corresponding program.

The port file can be viewed as a more versatile substitute for the Storage Queue mechanism.

Several declarations, statements, and features are used with ports.

- The file declaration PORT
- Extensions to OPEN, CLOSE, READ, WRITE and IF
- The statements GET, SET, and WAIT
- Port file attributes
- Function output parameters

These features are presented in this section. For further explanations, refer to the *BNA Architectural Description Reference Manual, Volume 1,* and to the *V Series MCP/VS Programming Reference Manual.*

CLOSE

The CLOSE statement terminates processing on the subfile specified in the PORT_KEY (refer to Function Output Parameters in this section), or all subports if the PORT_KEY equals zero. Figure 8-1 describes the format of the CLOSE statement.

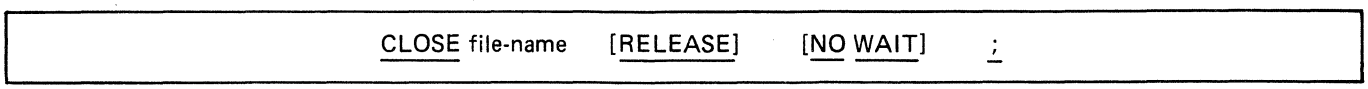


Figure 8-1. Format of the CLOSE Statement

File-name must be the name of a port file declared in a PORT statement.

If RELEASE is not specified, RETAIN is assumed.

If NO WAIT is not specified, the program will be suspended until the close is successful. When NO WAIT is specified, the program is reinstated immediately.

GET

The GET statement is used to obtain the current value of any attribute of a port file or a particular subfile of a port file. Figure 8-2 describes the format of the GET statement.

```
GET file-name identifier-1 FROM file-attribute-1

  [ , identifier-n  FROM  file-attribute-n]

 [RESULTS identifier-o]     [ [err-label] ]     ;
```

Figure 8-2. Format of the GET Statement

File-name must be the name of a port file declared in a PORT statement.

Identifier-1 through identifier-n specify the location to which the value of the attribute will be returned. The type and size of the identifier must correspond with the value of the attribute. An integer identifier must be used for Boolean and mnemonic attributes.

File-attribute-1 through file-attribute-n specify the file attribute identifier of the file attributes to be queried.

The RESULTS clause is used to obtain information on the results of the GET request. Identifier-o is required and must be previously declared. It should be an INTEGER with a size equal to two times the number of attributes in the list or an INTEGER ARRAY with an element size of two digits and one element for each attribute in the list. The MCP will return a 2-digit entry for each attribute in the list. Each entry will contain the error value resulting from the attempt to get the corresponding attribute. An error value of zero means no error. For additional error values, see the *BNA Architectural Description Reference Manual, Volume 1.*

The err-label provides an address to which program control will return if an error occurs in attempting to GET any attribute. If the err-label is not provided, no indication of an error will be given.

The user is responsible for updating the PORT_KEY with an appropriate subport index. (PORT_KEY is described under Function Output Parameters elsewhere in this section.) The values of the subport attributes listed will be returned for the specified subport. If the PORT_KEY is zero when querying subport attributes, a run time error will result. The PORT_KEY must be zero to return the values of port attributes.

A maximum of 99 attributes can be queried in a single statement.

Examples

```
GET     PORT01
        COUNT   FROM   CENSUS,
        CHG_EVENT   FROM   CHANGEEVENT;

GET     PORT02
        IN_FLAG   INPUTEVENT
        RESULTS   LSTERR    [ ER_LAB ] ;
```

IF

The IF statement can (optionally) interrogate the value returned for a mnemonic attribute that had been assigned to identifier-1 with the GET statement. The format is as follows:

```
IF [NOT] identifier-1 relational-operator mnemonic-attribute-value

    THEN statement-1    [ELSE statement-2]    ;
```

Figure 8-3. Format of IF Interrogating Identifier-1

Identifier-1 corresponds to identifier-1 in the GET statement. The list of valid mnemonic-attribute-values can be found in the port/subport attribute paragraphs. In this specific context, the mnemonic-attribute-values are treated as keywords with the exception of the words EXTERNAL, FIXED and IO which are always reserved words. In any other context, these keywords may be used as identifiers.

The list of possible mnemonic-attribute-values for each mnemonic attribute is represented internally as an ordered set of integers with ascending values. This allows all relational operators to be used when interrogating identifier-1. For example, for subport attribute FILESTATE, the value OPENED is considered greater than the value OFFERED.

This representation also allows identifier-1 to be used as the case selector in a CASE statement.

Mnemonic attributes with boolean values can be interrogated using the method described for the IF statement, option 3, identifier-5.

Examples

```
IF  ATTR_VAL  EQL  CLOSE  THEN
     GO  OPEN_IT;
IF  ATTR_VAL  GTR  NO_ERROR THEN
     GO  TO  ERROR_ROUTINE;
IF  INPUTEVENT  THEN
     GO  GET_IT;
```

OPEN

The OPEN statement is used to attempt to open the subfile specified in the PORT_KEY or all subfiles if the PORT_KEY equals zero. (PORT_KEY is described under Function Output Parameters, elsewhere in this section.) Figure 8-4 describes the format of the OPEN statement.

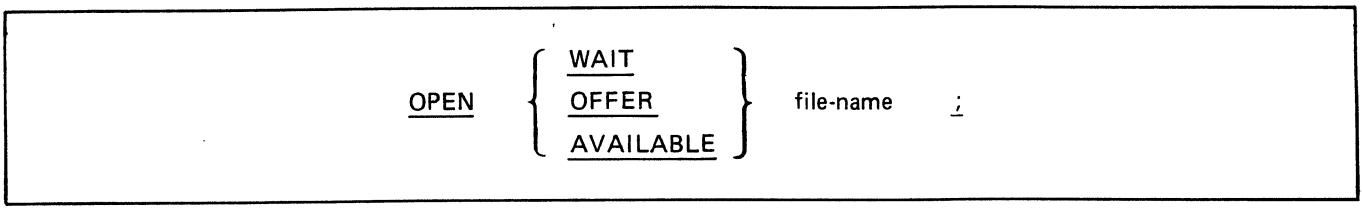


Figure 8-4. Format of the OPEN Statement

File-name must be the name of a port file declared in a PORT statement.

The WAIT option offers the subfile for matching. The program is suspended until a match is found.

The OFFER option offers the subfile for matching while the program continues to run.

The AVAILABLE option opens the subfile if a match is currently available. The program continues to run.

PORT

The PORT declaration is used to name the port file and to establish port and subport attributes. Figure 8-5 describes the format of the PORT declaration.

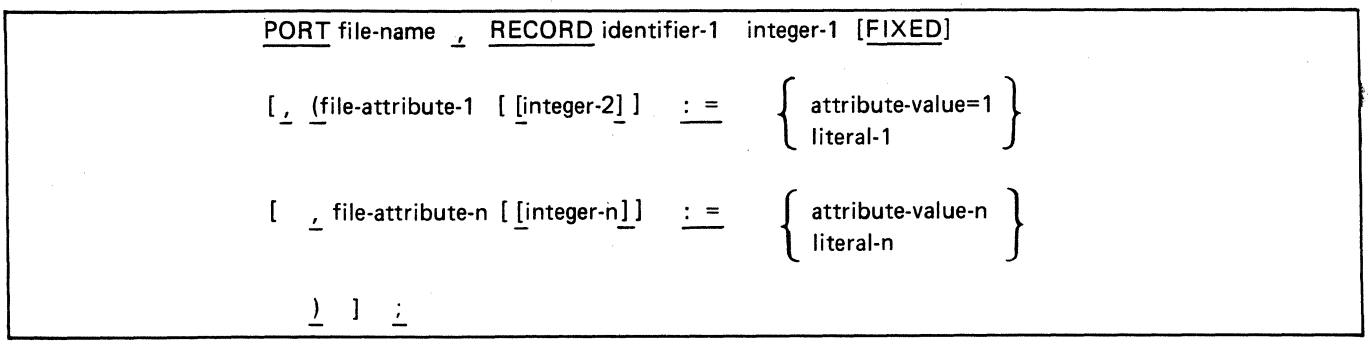


Figure 8-5. Format of the PORT Declaration

The file-name is used to identify the port for all file handling statements (READ, WRITE, etc.). It must be unique in the first seventeen characters.

The RECORD clause is required and must be followed by an identifier and an integer which specify the record name and record size, respectively, for this file. The data type of the record is ALPHA by default. Integer-1 must be six digits or less. The optional word, FIXED, is used to indicate that the declared record and size will not vary throughout the program. This allows the MCP to optimize the I/O path.

The record and/or size can be varied with the READ and WRITE statement if they have not been declared as FIXED.

File-attribute-1 through file-attribute-n specify the file attribute names of the file attributes to be preinitialized for the port and its subfile(s).

Integer-2 through integer-n specify the subport to which the attribute applies. It is required for all subport attributes and must be four digits or less. A value of zero indicates that this attribute applies to all subports. This index is not allowed when specifying port attributes.

Attribute-value-1 through attribute-value-n specify a mnemonic value and may only be assigned to boolean and mnemonic file attributes. The mnemonic values are listed in column two of the attribute tables. The key words TRUE and FALSE are considered mnemonic values in the case of boolean file attributes.

Literal-1 through literal-n specify the value to be assigned to alphanumeric and numeric file attributes.

Each port file must have a value for the TITLE, INTNAME and MAXRECSIZE attributes. If the TITLE or INTNAME attributes are not declared, the file-name will be used. If the MAXRECSIZE attribute is not declared, integer-1 will be used.

Examples

```
PORT    PORT01,  RECORD PORT_REC 100 FIXED,
        (TITLE := "TASKING",
         MYNAME  := "MASTER",
         MAXSUBFILES := 2,
         HOSTNAME [0]  := "SYSTEM1",
         YOURNAME [1]  := "SLAVE1",
         YOURNAME [2]  := "SLAVE2");

PORT    PORT2,  RECORD UTIL_REC 1000,
        (TITLE  := "UTILITY", MAXRECSIZE  := 1000);
```

READ

The READ statement is used to request the next message from a subport. Figure 8-6 describes the format of the READ statement.

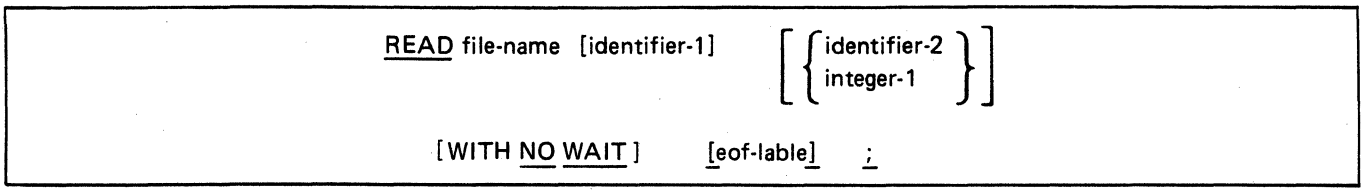


Figure 8-6. Format of the READ Statement

An OPEN statement must be successfully executed for a port file prior to the execution of the first READ statement for that file.

File-name must be the name of a port file declared in a PORT statement.

Identifier-1 and identifier-2 or integer-1 can only be used if the record name and size were not declared as FIXED for this file in the PORT statement. The record name or size may be specified independently. The use of identifier-1 changes the record address for this and all subsequent READ operations on the specified file. Identifier-1 must be ALPHA. The use of identifier-2 or integer-1 changes the record size for this and all subsequent READ operations on the specified file. Identifier-2 must be an INTEGER with a maximum size of six digits. The size specified need not correspond to the declared size of the specified record.

A READ statement causes the program to wait until a logical record is available. This suspension can be avoided for port files by specifying the WITH NO WAIT phrase.

The user is responsible for updating the PORT_KEY with an appropriate subfile index. (PORT_KEY is described under Function Output Parameter, elsewhere in this section.) If the PORT_KEY is non-zero, a read from the specified subfile is performed. If the PORT_KEY is zero, a non-directed read is performed and the PORT_KEY is updated to indicate the index of the subfile that was read.

The eof-label is used for numerous error conditions that are documented in the *BNA Architectural Description Reference Manual, Volume 1*. See also the *V Series MCP/VS Programming Reference Manual.*

Examples

```
READ     PORT01;
READ     PORT02 NEW_REC  200  NO WAIT;
READ     PORT03 50   [EOF];
```

SET

The SET statement is used to set one or more attributes of a port file or a particular subport of a port file. Figure 8-7 describes the format of the SET statement.

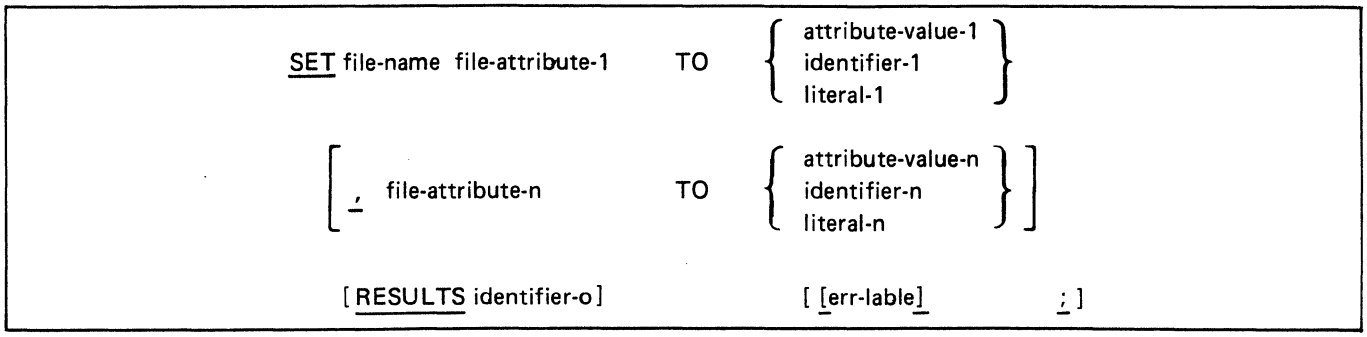


Figure 8-7. Format of the SET Statement

File-name must be the name of a port file declared in a PORT statement.

File-attribute-1 through file-attribute-n specify the file attribute identifier of the file attributes to be set for the port or subport.

Attribute-value-1 through attribute-value-n specify a mnemonic value and may only be used to set boolean and mnemonic file attributes.

Identifier-1 through identifier-n and literal-1 through literal-n specify the value to be assigned to an alphanumeric or numeric file attribute. The value must correspond to the range of the attribute being set.

The RESULTS clause is used to obtain information on the results of the SET request. Identifier-o is required and must be previously declared. It should be an INTEGER with a size equal to two times the number of attributes to be set or an INTEGER ARRAY with an element size of two digits and one element for each attribute to be set. The MCP will return a two digit entry for each attribute in the list. Each entry will contain the error value resulting from the attempt to set the corresponding attribute. An error value of zero means no error. For additional error values, see the *BNA Architectural Description Reference Manual, Volume 1.* See also the *V Series MCP/VS Programming Reference Manual.*

The err-label provides an address to which program control will return if an error occurs in attempting to set any attribute. If the err-label is not provided, no indication of an error will be given.

The user is responsible for updating the PORT_KEY with an appropriate subport index. (PORT_KEY is discussed under Function Output Parameters, elsewhere in this section.) If the PORT_KEY is nonzero, the subport attributes listed will be set for all subports. The PORT_KEY must be zero to set port attributes.

The user is also responsible for insuring that the file is in the proper state before attempting to set an attribute. The MCP requires all subports to be closed before any port attribute can be set.

A maximum of 99 attributes can be set in a single statement.

Example

```
SET     PORT01     MAXSUBPORTS  TO  5;

SET     PORT02     BLOCKSTRUCTURE FIXED,
                   SECURITYTYPE PRIVATE [ERROR];

SET     PORT03     HOSTNAME  TO  H_NAME
      RESULTS      ERRLST;
```

WAIT

The WAIT statement is used to cause the suspension of the execution of the object program for a specified interval of time or until one or more conditions are true. Figure 8-8 describes the format of the WAIT statement.

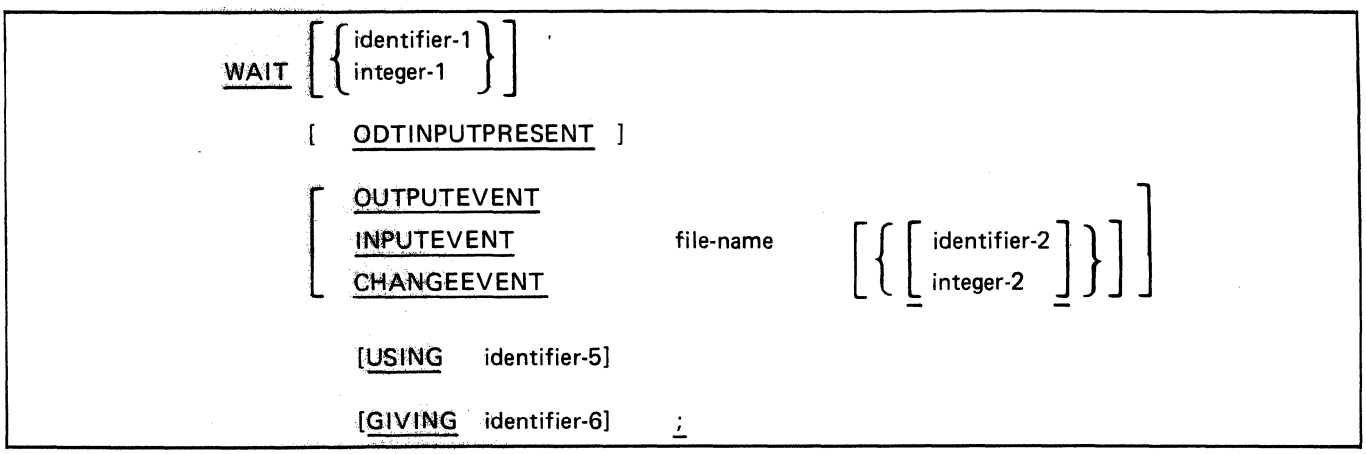


Figure 8-8. Format of the WAIT Statement

A WAIT with no options specified causes the program to be suspended for 86400 seconds (24 hours).

Not more than one identifier-1 or integer-1 can be specified, and if specified, must be the first item in the list. Identifier-1 must be an INTEGER of five digits or less. The maximum value for identifier-1 and integer-1 is 86399.

When ODTINPUTPRESENT is specified, the execution of the object program is suspended until an AX MCP Keyboard input message is received for the program.

When OUTPUTEVENT, INPUTEVENT or CHANGEEVENT is specified, the program is suspended until the specified port file event becomes TRUE. File-name must be the name of a file declared in a PORT statement. Identifier-2 and integer-2 specify the index of the subport to be checked. Identifier-2 must be an INTEGER of four digits or less. If identifier-2 or integer-2 is omitted or has a value of zero, the port file is checked. OUTPUTEVENT is only valid for a subport.

If any event in the list is TRUE, the WAIT is terminated and control is passed to the next executable statement.

If during the initial scan of the events, no event is found to be TRUE, program execution is suspended until any event in the list becomes TRUE. At that time, the WAIT statement is terminated and control is passed to the next executable statement.

When the USING clause in not specified, each event in the list is tested for a TRUE condition, beginning with the first event in the list and proceeding to the last event in the list. The process then repeats. When the USING clauseis specified, the value referenced by indentifier-5 determines where in the specified event list the WAIT should begin testing for a TRUE condition. Identifier-5 must be an INTEGER of two digits.

When the GIVING clause is specified, the data item referenced by identifier-6 is set to the position in the event list of the event that terminated the WAIT statement. Identifier-6 must be an INTEGER of two digits.

Examples

```
WAIT   50;
WAIT   MAX_TIME
       ODTINPUTPRESENT;

WAIT   MAX_TIME
       OUTPUTEVENT          PORT01   [ 1 ]
       CHANGEEVENT          PORT01
       INPUTEVENT           PORT02   [ SUB_INDEX ]
       INPUTEVENT           PORT02   [ 3 ]
       CHANGEEVENT          PORT02
       USING                POINTER
       GIVING               RESULT;
```

WRITE

The WRITE statement is used to request that a message be sent through a subport. Figure 8-9 describes the format of the WRITE statement.

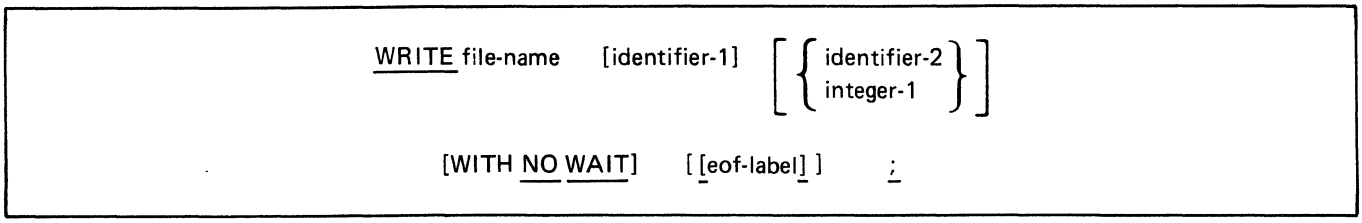


Figure 8-9. Format of the WRITE Statement

An OPEN statement must be successfully executed for a file prior to the execution of the first WRITE statement for that file.

File-name must be the name of a port file declared in a PORT statement.

Identifier-1 and identifier-2 or integer-1 can only be used if the record name and size were not declared as FIXED for the file in the PORT statement. The record name or size may be specified independently. The use of identifier-1 changes the record address for this and all subsequent WRITE operations on the specified file. Identifier-1 must be ALPHA. The use of identifier-2 or integer-1 changes the record size for this and all subsequent WRITE operations on the specified file. Identifier-2 must be an INTEGER with a maximim size of six digits. The size specified need not correspond to the declared size of the specified record.

A WRITE statement causes the program to wait until a buffer is available to store the record. This suspension can be avoided for port files by specifying the WITH NO WAIT phrase.

The user is responsible for updating the PORT_KEY with an appropriate subport index. (PORT_KEY is described under Function Ouptut Parameters, elsewhere in this section.) If the PORT_KEY is nonzero, a write to the specified subport is performed. If the PORT_KEY is zero, a broadcast write is performed, for which data are sent to all opened subport of the port file.

The eof-label is used for numerous error conditions which are documented in the *BNA Architectural Description Reference Manual, Volume 1.*

Examples

```
WRITE       PORT01;

WRITE       PORT02   PORT_DATA   500   [ EOF ] ;

WRITE       PORT03   1000   NO   WAIT;
```

PORT FILE ATTRIBUTES

Port file attributes allow a user to request the compiler to generate code to directly manipulate or interrogate the characteristics of a port file.

A file attribute may be alphanumeric, numeric or mnemonic. An alphanumeric port file attribute is considered an ALPHA data item having a size equal to the mazimum size allowed for the specified attribute. Its content is left-justified and blank filled. A numeric port file attribute is considered an INTEGER data item having a size equal to the mazimum size allowed for the specified attribute. A mnemonic port file attribute is associated with values that are best expressed as mnemonic names. Port file attributes having a boolean nature are considered mnemonic attributes and are associated with the values TRUE and FALSE.

Table 8-1 lists the valid port attributes; table 8-2 lists the subports attributes. The columns of these tables are defined as follows:

Column 1 -	File attribute name
Column 2 -	Attribute value
Column 3 -	Access restrictions 1 - Attribute may be specified in a PORT declaration 2 - Attribute may be interrogated in a GET statement 3 - Attribute may be specified in a SET statement
Column 4 -	When attribute can be accessed in a GET statement A - anytime
Column 5 -	When attribute can be accessed in a SET statement A - anytime C - all subports must be closed for port attributes; the specified subport must be closed for subports attributes

Table 8-1. Port Attributes

Name	Values	Access	Get	Set	Type	Default
BLOCKSTRUCTURE	FIXED EXTERNAL	1,2,3	A	C	M	FIXED
CENSUS	0 THRU 99999	2	A	-	N	0
CHANGEEVENT	TRUE, FALSE	2	A	-	M	
CHANGEDSUBFILE	0 THRU 9999	2	A	-	N	0
INPUTEVENT	TRUE, FALSE	2	A	-	M	
INTNAME	ALPHA (17)	1,2,3	A	C	A	NOTE 1
LASTSUBFILE	0 THRU 9999	2	A	-	N	0
MAXRESCIZE	2 THRU 19998 (BYTES)	1,2,3	A	C	N	NOTE 2
MAXSUBFILES	1 THRU 9999	1,2,3	A	C	N	1,6
MYHOSTNAME	ALPHA (17)	2	A	-	A	NOTE 3
MYNAME	ALPHA (99)	1,2,3	A	C	A	NULL
SECURITYGUARD	ALPHA (99)	1,2,3	A	C	A	NULL
SECURITYTYPE	PUBLIC PRIVATE GUARDED	1,2,3	A	C	M	NOTE 4
SECURITYUSE	I0	1,2,3	A	A	M	IO
TITLE	ALPHA(17)	1,2,3	A	C	A	NOTE 5

Notes:

1. If not declared in the PORT declaration, the file-name is used.
2. If not declared in the PORT declaration, the record size is used.
3. The local host name is used.
4. If not declared in the PORT declaration, the SECURITYTPE is obtained from the user's session when the program is executed.
5. If not declared in the PORT declaration, the file-name is used.
6. Under MCP/VS 2.0, MAXSUBFILES is temporarily restricted to a maximum of 700.

Table 8-2. Subfile Attributes

Name	Values	Access	Get	Set	Type	Default
CENSUS	0 THRU 9999	2	A	-	N	0
CHANGEEVENT	TRUE, FALSE	2	A	-	M	
COMPRESSION	TRUE, FALSE	1,2, 3	A	*	M	FALSE
COMPRESSIONPOSSIBLE	TRUE, FALSE	2		-	M	FALSE
CURRENTRECORD	2 THRU 19998 (BYTES)	2		-	N	
DIALOGPROTOCOLLEVENT	0 THRU 255	2		-	N	0
FILESTATE	CLOSED AWAITING_HOST OFFERED OPENED SHUTTING_DOWN BLOCKED CLOSED_PENDING DEACTIVATION_PENDING DEACTIVATED	2		-	M	CLOSED
HOSTNAME	ALPHA (17)	1,2, 3	A	C	A	NOTE 1
INPUTEVENT	TRUE, FALSE	2	A	-	M	
MAXCENSUS	0 THRU 9999	1,2, 3	A	A	N	3
MAXRECSIZE	2 THRU 19998 (BYTES)	2	A	-	N	
OUTPUTEVENT	TRUE, FALSE	2	A	-	M	
SUBFILEERROR	NO_ERROR DISCONNECTED DATA_LOST NO_BUFFER NO_FILE_FOUND UNREACHABLE_HOST UNSUPPORTED_FUNCTION	2	A	-	M	NO_ERROR
YOURNAME	ALPHA (99)	1,2, 3	A	C	A	NULL
YOURUSERCODE	ALPHA (17)	1,2, 3	A	C	A	NOTE 2

* When COMPRESSIONPOSSIBLE = TRUE

Notes:

1. If not declared in the PORT declaration, the local host name is used.
2. If not declared in the PORT declaration, the value for YOURUSERCODE is obtained from the user's session when the program is executed.

FUNCTION OUTPUT PARAMETERS

Ten fields in each port file FIB are updated by the MCP after each READ and WRITE request on the file. One of the fields is also used for input information. These fields can be accessed by the user through the reserved words described below.

PORT_KEY - 4 UN

The PORT_KEY field must be set before each request on a port file, if more than one subport is associated with the file. This field specifies the subport index for the request. A value of zero means 'all subports' or 'port' depending on the type of request. After a successful non-directed read, this field will be set to the index of the subport returning the data.

PORT_MAXMSG - 6 UN

This field specifies the maximum possible data size, in bytes, on the last I/O request.

PORT_ERROR - 2 UN

This field specifies the error value from the last request. It is also updated after an OPEN and CLOSE. (see note)

PORT_STATE - 1 UN

This field specifies the subport state at the completion of the last request. It is also updated after an OPEN and CLOSE. (see note)

PORT_EOF - 1 UN

This field specifies the end-of-file condition as of the completion of the last read or write request. (see note)

PORT_Q - 6 UN

This field specifies the total number of input messages remaining to be read in all subports as of the completion of the last request.

PORT_SUBQ - 4 UN

This field specifies the number of input messages remaining to be read from the subports specified in the PORT_KEY field as of the completion of the last request.

PORT_CURREC - 6 UN

This field specifies the size, in bytes, of the current record as of the completion of the last read of write request.

PORT_DATE - 5 UN

This field specifies the Julian date as of the completion of the last READ request.

PORT_TIME - 10 UN

This field specifies the time of day in milliseconds as of the completion of the last READ request.

NOTE
For a list of all possible values, see the *BNA Architectural Description Reference Manual, Volume 1.*

These reserved words are intended to be used as overrides on port file name as follows:

file-name.override

Their primary use is in assignment and IF statements to set or interrogate the value of a particular function output parameter.

Examples

```
PORT01.PORT_KEY := 1;

STATE1 := PORT01.PORT_STATE;

IF  PORT01.PORT_ERROR GTR 0 THEN
    GO HANDLE_ERROR;
```

SECTION 9
READER SORTER - PRE-4A CONTROL CONSTRUCTS

GENERAL

This section deals with the BPL constructs required to activate the READER SORTER equipment as defined by the FILE declaration SORTER clause.

READER SORTER FILE HANDLING

Certain reader/sorter file functions must be performed in use routines. The routine types and their functions are listed in Table 9-1. Refer to the ROUTINE clause of the FILE declaration for further details.

Table 9-1. Routine Types and Their Functions

Routine Type	Function
LABEL	Identifies the routine to process memory access, cannot read, unencoded, and double document errors.
IOERROR	Amount field error procedure.
EOP	Transit field error procedure.
SORTER	Item pocket-select routine.

The reader sorter file must be declared RANDOM.

The KEY clause of the FILE declaration specifies the label of the routine to handle manual end-of-file, jam, and mis-sort conditions.

SPECIFIC STATEMENT FORMATS

The specific statement formats together with a detailed discussion of the restrictions and limitations associated with each, appear on the following pages in alphabetic sequence.

NOTE

The use of any of the following requires the presence of the MICR module of the MCP.

ACTION 0 (Pocket Select)

The function of ACTION 0 is to pocket select the last document read to the pocket specified on the READER SORTER. Figure 9-1 describes the format of ACTION 0.

```
ACTION file-name 0 data-name-1 [ too-late-to-process-label ] ;
```

Figure 9-1. Format of ACTION 0

Data-name must be declared as a four digit field and its format is NNRV, where: NN is the pocket to be selected, R is zero, and V is either zero (if the current mode is to continue) or one (if FLOW is to be stopped).

If the ACTION 0 for the document was too late to process, the program will branch to the too-late-to-process-label. FLOW mode is stopped and the document has been sent to the reject (R) pocket. The information read from the document which caused the too-late-to-process is stored in the record area. However, the trailing documents will not have been placed in memory, but will be routed to the reject (R) pocket.

ACTION 4 (Pocket Light)

The function of this ACTION is to cause a specified READER SORTER pocket light to become illuminated. Figure 9-2 describes the format of ACTION 4.

```
ACTION file-name 4 data-name-1 ;
```

Figure 9-2. Format of ACTION 4

Data-name-1 must be declared as a two (2) digit field.

Data-name-1 must contain the 2-digit pocket number which specifies the pocket light desired to be turned "ON".

Flow must be stopped and all documents pocket selected before issuing an ACTION 4 statement.

Control is set to a NOT READY condition and must be cleared by depressing the START button on the READER SORTER.

ACTION 6 (Batch Count)

The function of this ACTION is to increment the batch counter in the READER SORTER by one. Figure 9-3 describes the format of ACTION 6.

```
ACTION file-name 6 ;
```

Figure 9-3. Format of ACTION 6

Flow must be stopped and all documents pocket selected before issuing an ACTION 6 statement.

ACTION 8 (Delay)

The function of this ACTION is to delay the start of reading characters. Figure 9-4 describes the format of ACTION 8.

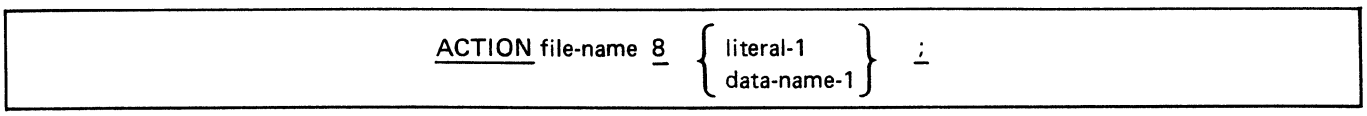


Figure 9-4. Format of ACTION 8

This statement sets a timer to the number of seconds to delay the start, specified by literal-1 or data-name-1.

Literal-1 must be a 1 to 4 digit literal.

Data-name-1 must be a declared INTEGER (4).

OPEN

The function of this statement is to initiate the input processing of the READER SORTER. Figure 9-5 describes the format of OPEN.

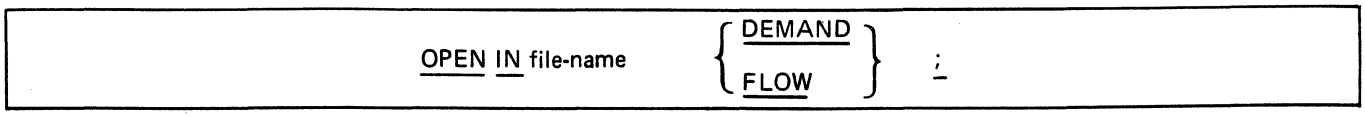


Figure 9-5. Format of OPEN

At least one of the options must be specified before a file can be read. A CLOSE statement is required if it becomes necessary to change the mode of operation from either OPEN in DEMAND or FLOW mode.

READ

The function of this statement is to make available the next logical record from the READER SORTER in DEMAND or FLOW mode. Figure 9-6 describes the format of READ.

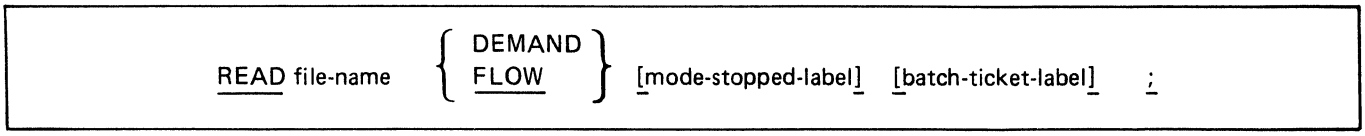


Figure 9-6. Format of READ

The record size must be declared as 200 characters.

The READ FLOW . . . . statement must be given after an OPEN IN FLOW. This will start the flow of documents through the READER SORTER.

If CHECK 4 is specified in the FILE declaration, the data is stored (in descending sequence) continuously. If formatting is specified (CHECK 4 is omitted), the data is stored (in descending sequence) continuously until the first transit symbol is received. Blanks are then stored until the 40th character location is reached at which point the transit symbol and remaining data is stored. Blanks are stored following the last information character read until a total of 100 characters is stored. When formatting (CHECK 4 is omitted) is specified, automatic validity checking of the amount and transit fields is performed. Validity checking of the amount field includes checking:

- The first and twelfth characters stored for amount symbols
- The 10 interventing characters for decimal digits

Validity checking of the transit field includes checking:

- The 40th and 50th characters stored for transit symbols
- The nine intervening characters for the following: four decimal digits, hyphen (-), and four decimal digits

Mode-stopped-label entry specifies the procedure to be executed when the flow mode is stopped. All documents which were in motion will be processed and pocket selected before going to mode-stopped-label. A READ FLOW statement has to be executed to restart the READER SORTER in a flow mode.

Batch-ticket-label entry specifies the procedure to be executed if a batch ticket (black band) was encountered during the last document pocket selection. The record area has been blanked. The READER SORTER is in a stop flow mode and must be restarted with a READ FLOW statement.

SECTION 10
READER SORTER - DLP/4A CONTROL CONSTRUCTS

GENERAL

This section deals with the BPL constructs required to activate the READER SORTER equipment connected to the system through a DLP (900-series systems) or through a 4A I/O Control. In either of these cases, the hardware name in the FILE declaration is SORTER4.

Refer to the *B 2000/B 3000/B 4000 Series MCP Programmer's Guide* for detailed information on the Reader/Sorter DLP and 4A Control application program interfaces.

READER SORTER FILE HANDLING

Certain reader/sorter file functions must be performed in use routines. The routine types and their functions are listed in Table 10-1. Refer to the ROUTINE clause of the FILE declaration for further details.

Table 10-1. Routine Types and Their Functions

Routine Type	Function
LABEL	Identifies the routine to process memory access, Cannot Read, unencoded, and double document errors.
IOERROR	Amount field error procedure.
EOP	Transit field error procedure.
SORTER	Item pocket-select routine.

The reader sorter file must be declared RANDOM.

The KEY clause of the FILE declaration specifies the label of the routine to handler manual end-of-file, jam, and mis-sort conditions.

SPECIFIC STATEMENT FORMATS

The specific statement formats together with a detailed discussion of the restrictions and limitations associated with each, appear on the following pages in alphabetic sequence.

NOTE

The use of any of the following requires the presence of the MICR module of the MCP.

ACTION 10 (Pocket Select)

The function of this statement is to exit from the POCKET SELECT USER routine. Figure 10-1 describes the format of ACTION 10.

```
ACTION file-name 10 ;
```

Figure 10-1. Format of ACTION 10

An ACTION 10 statement must be issued by the POCKET SELECT USER ROUTINE for each item. This implies that the user routine is ready to read and pocket select the next item as soon as the character recognition system and the control make it available.

ACTION 11 (Pocket Light Generate)

The function of this statement is to illuminate the specified pocket light or generate image count marks on microfilm. Figure 10-2 describes the format of ACTION 11.

```
ACTION file-name 11 [error-label] ;
```

Figure 10-2. Format of ACTION 11

Error-label is the branch taken whenever an error is detected in execution of this statement.

ACTION 12 (Status Inquiry)

The function of this statement is to obtain information regarding the status of the READER SORTER. Figure 10-3 describes the format of ACTION 12.

```
ACTION file-name 12 [error-label]  ;
```

Figure 10-3. Format of ACTION 12

The MCP places the status of the Reader Sorter in program soft interface area (buffer location 757).

The status is a 1-digit field. The formatting of the status digit is:

LOCATION 757:

Bit 8	= 1 : Slewing microfilm.
Bit 4	= 1 : Camera not ready.
	= 0 : Camera ready or not present.
Bit 2	= 1 : Endorser not ready.
	= 0 : Endorser ready or not present.
Bit 1	= 1 : Sorter not .ready.

ACTION 13 (Charateristics Inquiry)

The function of this statement is to obtain the characteristics of the READER SORTER. Figure 10-4 describes the format of ACTION 13.

```
ACTION file-name 13 [error-label]  ;
```

Figure 10-4. Format of ACTION 13

The MCP places the characteristics of the READER SORTER in the program soft interface area (buffer location 758-759).

The READER SORTER characteristics are formatted in this 2 digit field, as follows:

LOCATION 758:

Bit 8 = 1	: Endorser band one present.
Bit 4 = 1	: Endorser band two present.
Bit 2 = 1	: Endorser band three present.
Bit 1 = 1	: Endorser band four present.

LOCATION 759:

Bit 8 = 1	: Reader sorter is a B 9137-2.
Bit 4 = 1	: Camera present.
Bit 2	: Reserved.
Bit 1 = 1	: Read station B present.

ACTION 14 (Microfilm Advance)

The function of this statement is to advance the microfilm to the beginning of the next 100-foot or 200-foot reel within a cassette. Figure 10-5 describes the format of ACTION 14.

```
ACTION file-name 14 [error-label]  ;
```

Figure 10-5. Format of ACTION 14

ACTION 15 (Start Flow)

The function of this command is to initiate flow feed. Figure 10-6 describes its format.

```
ACTION file-name 15  [error-label]  [stopped-label]  ;
```

Figure 10-6. Format of ACTION 15

Just prior to flow feed, the control reads the soft loaded delimiters, read delay, stop information, and band information from system memory. While in flow mode, only the POCKET SELECT and READ READER SORTER commands may be initiated.

Flow must be stopped prior to issuing a start flow.

Since start is not treated as a logical read, the user program must proceed to a READ READER SORTER command when it has been reinstated.

If flow cannot be successfully started due to an error condition, the user program is reinstated at the error-label as specified at error-label.

If flow is stopped, the stopped-label branch is taken.

CLOSE

The function of this statement is to close a READER SORTER file. Figure 10-7 descibes the format of CLOSE.

```
CLOSE file-name RELEASE ;
```

Figure 10-7. Format of CLOSE

A 4A-type Reader Sorter file must to closed with RELEASE.

OPEN

The function of this statement is to initiate the input processing of the READER SORTER. Figure 10-8 describes the format of OPEN.

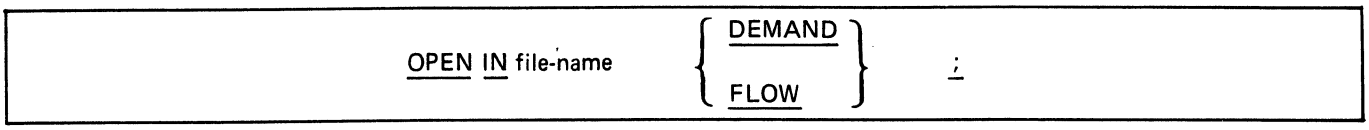


Figure 10-8. Format of OPEN

At least one of the options must be specified before a file can be read. A CLOSE statement is required if it becomes necessary to change the mode of operation from either OPEN in DEMAND or FLOW mode.

READ

The function of this statement is to make available the next logical record from the READER SORTER in DEMAND or FLOW mode. Figure 10-9 describes the format of READ.

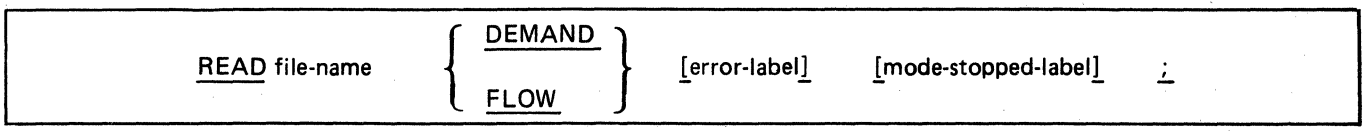


Figure 10-9. Format of READ

The record size must be declared as 780 characters.

The READ FLOW . . . . statement must be given after an OPEN IN FLOW. This will start the flow of documents through the READER SORTER.

Error-label entry specifies the procedure to be executed whenever an error is detected during the execution of this statement.

Mode-stopped-label entry specifies the procedure to be executed when the flow mode is stopped. All documents which were in motion will be processed and pocket selected before going to mode-stopped-label. A READ FLOW statement has to be executed to restart the READER SORTER in a flow mode.

BUFFER

When using SORTER4 constructs, the user must initialize the hard interface portion of the 780-digit buffer. The soft interface area is used by the MCP for control and for communicating to the user.

USER FILE STATEMENT

The users input/output and interface buffer is located at the address specified in FIB-WA. It contains 756 digits of hard interface, followed by 24 digits of soft interface, for a total record size of 780 digits. The interface areas are defined in the *B 2000/B 3000/ B 4000 Series MCP Programmer's Guide.*

SECTION 11
OPERATING INSTRUCTIONS

GENERAL

The BPL compiler, in conjunction with the MCP, allows various types of action during compilation. Control of the BPL source language input is derived from presenting the compilation file to the Master Control Program, and is explained as follows:

1. The first input control record notifies the MCP to call out the BPL compiler and to compile the indicated program-name. In the absence of this control record, the system operator may manually execute one of the compile options through the ODT.

2. The second control record is the label record, and is formatted in the following manner:

   - ? DATA CARD (indicates EBCDIC source language input).

   - ? DATAB CARD (indicates BCL source language input).

3. The third record is the compiler option control record ($ in column 1). This record is used to notify the compiler as to which options are required during compilation. If this record is omitted, $ CARD LIST is assumed.

The Dollar-sign ($) must be in column 1. Any record containing a dollar-sign in column 1 is considered to be a compiler option control record.

SET followed by one or more options sets the specified options (on) leaving all others unchanged.

RESET followed by one or more options resets the specified options (off) leaving all others unchanged.

If more than one option is requested on the same record for a function, the last option specified is accepted.

Any compiler option control record not specifying SET or RESET will turn off all options not specified on that record.

Allowable options are:

- *CARD* - input is from source language cards or paper tape only. The CARD option should not be used on the same control record as the TAPE, DISK, or PACK options.
- *NEWC* - creates a new source language card file (EBCDIC). The NEWC option should not be used on the same control record as the NEWT, NEWD, or NEWP options.
- *TAPE* - indicates the primary input is from a source language tape with a file-identifier of BPLSYM. The tape must contain 80-character card images with a blocking factor not exceeding nine. The TAPE option should not be used on the same control record as the CARD, DISK or PACK options.
- *NEWT* - creates a new source language tape file, including all pertinent changes, with a file-identifier of BPLSYM. Th BPLSYM tape file is created as 80-character record blocked name. The NEWT option should not be used on the same control record as the NEWC, NEWD, or NEWP options.
- *DISK* - indicates that the primary input is from a source language disk file with a file-identifier of BPLSYM, the BPLSYM disk file must contain 80-character card images, blocked five, nine, or multiples of five. The DISK option should not be used on the same control record as the CARD, TAPE, or PACK options.
- *NEWD* - creates a new source language disk file, including all pertinent changes, with a file-identifier of BPLSYM. The BPLSYM disk file is created as 80-character record, blocked nine. The NEWD option should not be used on the same control record as the NEWC, NEWT, or NEWP options.
- *PACK* - indicates to the compiler that the primary input is from a source language disk pack file with a file-identifier of BPLSYM. The BPLSYM disk pack file must contain 80-character card images, blocked five, nine, or multiples of five. The PACK option should not be used on the same control record as the CARD, TAPE, or DISK options.
- *NEWP* - creates a new source language disk pack file, including all pertinent changes, with a file-identifier of BPLSYM. The BPLSYM disk pack file is created as 80-character records, blocked nine. The NEWP option should not be used on the same control record as the NEWC, NEWT, or NEWD options.
- *LIST* - creates a full listing, double spaced, of the source language input with error messages where required.
- *LST1* - same as LIST with single spacing.
- *NPRT* - inhibits printing of compiler summary if the LIST and LST1 options are not specified. Syntax errors will always be listed regardless of the listing action specified.
- *LNXX* - the number of lines on a page for source printout may be specified by the programmer with this option. The letter XX designate the number of lines desired. If this option is omitted from the $ record, channel 12 is used to control page skipping.

- *JAPN* - the output listing will be compressed and will start in column 37 of the print line.
- *CODE* - list object code from the point of insertion.
- *SUPR* - suppresses warning messages.
- *SPEC* - negates LIST and LST1 if syntax errors occur.
- *BLNK* - causes all source file records with positions 1-72 blank to be automatically purged. A subsequent control record without BLNK will turn off this option.
- *RSEQ* NNNNNNNN +NNNNNNNN - indicates resequencing of source language input onto the output listing and/or new source language file is desired. The first integer is the starting sequence number, the second integer, preceded by a plus sign (+) is the increment factor. If either integer is less than eight digits, leading zeros are assumed. The default integers are 00000000 and +00001000. Either of the integers appearing without RSEQ will be ignored.
- *DLIS* - suppresses listing of dropped records in a $$ DROP conditional statement.
- *UPER* - lists in upper case for the printers without lower case. Symbolic is unchanged.
- *REMV* - removes any old copy of the object file and replaces it with the new copy.
- *VOID NNNNNNNN* - VOIDs symbolic records from the point of insertion *up to but not including* the sequence number indicated by NNNNNNNN. The sequence number is required.
- *MTCH* - matches BEGIN and END pairs on the program listing.
- *XREF* <memory size> - indicates that a cross reference of the compiled program is to be listed by the program BPLXRF. <memory size> is optional and initiates BPLXRF with the memory size stated. When used, this option must appear in the first leading dollar record; see information on leading dollar records, following.
- *FLAG NN* - SETS or RESETS FLAG numbers (NN) for conditional compiling. NN is a number from 01 to 50.
- *STK4* - will force all runtime stack frames to be aligned on mod-4 addresses.
- *ICM2* - causes creation of a Type II ICM file from this compilation, rather than executable code. When used, this option must appear on a leading dollar record (see following), and it cannot be reset. See Appendix D for further information.
- *ICM3* - causes creation of a Type III ICM file from this compilation, rather than executable code. When used, this option must appear on a leading dollar record (see following), and it cannot be reset. See Appendix D for further information.

NOTE

The ICM3 compiler option is only available with a BPL compiler in the 2.*n* and later series of releases. This option is not available in the 7.*n* series of BPL compilers.

BPL uses the sequence number field in the $ records contained in the symbolic file to determine which are leading $ records and which are imbedded. Any $ records with blank sequence numbers are leading $ records and any containing sequence numbers are imbedded, even if they should happen to be at the beginning of the symbolic file. (If the symbolic file is entered through CANDE, a $ record must be included at the beginning of the file to be considered a leading $ record.)

This is important because certain options are ignored unless their value is SET at the time the *last leading* dollar record is processed. These options are:

DISC	PACK	NEWC
DISK	NEWT	XREF*
CARD	NEWD	JAPN
TAPE	NEWP	ICM2
		ICM3

*Note:
XREF must be specified in the *FIRST* leading dollar record, or it is ignored.

An alternative to the steps specified on page 9-1 is to compile a BPL program by entering control information through the ODT as follows:

COMPILE . . . . INSERT 0 7 ABCDEFG

or,

COMPILE . . . . VALUE 0 = ABCDEF

Under MCP/VS 2.0 or later:

COMPILE <program-name> BPL (ABCDEFG0) LIB

Table 11-1 shows the meaning of each bit in the value statement.

Table 11-1. Bits in the Value Statement

Digit	Bit	Function
A	8	Patch input is on disk pack; file-identifier is CARD
	4	Patch input is on disk; file-identifier is CARD
	2	Patch input is on magnetic tape; file-identifier is CARD
	1	Patch input is on cards (default); file-identifier is CARD.
B	8	Master symbolic input is on disk pack (PACK)
	4	Master symbolic input is on disk (DISK)
	2	Master symbolic input is on magnetic tape (TAPE)
	1	Patch input is an EDITOR format file
C	8	Create new symbolic disk pack file (NEWP)
	4	Create new symbolic disk file (NEWD)
	2	Create new symbolic magnetic tape file (NEWT)
	1	Create new symbolic card file (NEWC)
D	8	Match begin/end pairs (MTCH)
	4	Delete blank records (BLNK)
	2	List cross reference (XREF)
	1	Reserved
E	8	List generated code (CODE)
	4	List symbolic input, double spaced (LIST)
	2	List symbolic input, single spaced (LST1)
	1	Suppress warning messages (SUPR)
F	8	List symbolic input beginning in print position 37 (JAPN)
	4	Print listing using upper case only (UPER)
	2	List only error lines (NPRT)
	1	Negate LIST and LST1 if syntax errors occur (SPEC)
G	8	Replace an old copy of object file with new copy (REMV)
	4	Reserved
	2	Suppress listing of dropped records in $$DROP statement (DLIS)
	1	Reserved.

Example

COMPILE EXAM WITH BPL VALUE 0 = 240600

1. Compile BPL program EXAM
2. Patch input is a magnetic tape
3. Master Program is a magnetic tape
4. No new file
5. Delete blank records
6. Cross-reference list

COMPILE EXAM WITH BPL INSERT 0 7 0002480

1. Compile BPL program EXAM
2. Patch input is cards (default)
3. Cross-reference listing
4. Double space list
5. 37 print position printing (compressed listing)

Compiler Operational References

Occasionally, a compile-time address error will occur due to stack overflow. This happens, for example, when IF or CASE statement are nested too deeply or there are too many levels in a data description for the compiler to handle with its normal stack mechanism.

When this happens, the stack size can be increased beyond the 3000-digit default by inserting into the ?COMPILE ... MCP control record, INSERT 10 6 NNNNNN where NNNNNN is a 6-digit number larger than 003000, specifying total stack size in digits. An MCP MEMORY clause must also be included to increase the compiler size by an equal number of digits.

Large programs require additional memory for efficient compiles. In general, any program with more than 2500 lines of source will compile more efficiently if the compiler is given more memory. Increments of 10KD should be added as the size of the source increases. In addition, the following errors may occur during a compile which can indicate insufficient memory allocation.

- Erroneous syntax errors 2311, and 2601.
- Compiler failure with invalid arithmetic data.

File Equate Information

When you run the BPL compiler in a multiprogramming environment, you need to know the internal and external names BPL uses for its input and output files. Table 11-2 provides a list of these names.

Table 11-2. Internal and External I/O File Names

File	Internal Name	External Name
Master symbolic in (CARD, TAPE, DISK, PACK).	MSTRFL	BPLSYM
Patch symbolic in (CARD, TAPE, DISK, PACK).	PTCHFL	CARD
Patch symbolic in (Editor format).	REMTFL	CARD
Master symbolic out (CARD, TAPE, DISK, PACK).	MSTROT	BPLSYM
Symbolic/error list (PRINTER).	PRINT	PRINT

Example

```
? FILE MSTROT = FORTOT

? FILE PRINT = FORTPR
```

Input

The compiler can merge inputs from two sources on the basis of sequence numbers. When inputs are merged, the listing indicates inserts by a "*" preceding the sequence number, and replacementsby a "#" preceding #the sequence number.

CANDE Editor Format Files

CANDE EDITOR format files can be used as input to a BPL compiler and can function as patch input to a disk, tape, or disk pack symbolic. However, new symbolics created while using EDITOR files will not be in EDITOR file format. EDITOR input can be used for both batch and timeshared compiles. If running under the timeshared environment, the EDITOR will handle the necessary communication with the compiler. If running in batch, however, address 1 of the compiler must be set to 1 to indicate EDITOR format input; this is done with a VALUE or INSERT command in the COMPILE statement. For example, VA 0=410000 states that an EDITOR patch file on disk will be used; VA 0=810000 indicates a patch file on diskpack. Other dollar record options can also be included in this VALUE or INSERT statement.

BPL compiles require either a leading dollar record (blank sequence number) as the first symbolic or a VALUE 0 or INSERT 0 statement if options other than the default values of card patch, double spaced listings are desired. However, since EDITOR format files require sequence numbers, the first symbolic record in an EDITOR files will be treated as a leading dollar record if: (1) a dollar sign ($) appears in the first position, (2) a dollar sign ($) does not appear in the second position ("$$ DROP..." record is not a dollar record), and (3) the words "SET" or "RESET" do not appear as the next character string on that dollar record. Only the first record will be treated in this manner, and the grouping of several such dollar records at the beginning of the file will be the equivalent to having one leading dollar record and several other dollar records which reset the previous dollar record values.

When running in the timeshared environment, a syntax error display listing a maximum of twenty errors and the associated symbolic lines will automatically be displayed onto the remote device from which the compile was initiated. In addition, any requested printer format listing will be generated.

APPENDIX A
BPL RESERVED AND KEY WORDS

BPL places reserved type words in two levels, Class I (RESERVED) and Class II (KEY).

A Reserved word is reserved throughout the entire compiler. It may be DEFINEd to give it new meaning within a block structure or the entire program.

A Key word is only reserved in those contexts in which it is meaningful. Any other usage is permitted within the compiler. For instance the following example will function properly:

```
INTEGER MVC (12) MOD 4, B(12);
B := [MVC] MVC;
```

Reserved Words - Class 1

ACCEPT
ACCESS
ACTION
ADDRESS
ALPHA
AND
ARM
ARRAY
ATT8A1
AVAILABLE

BASE
BCT
BEGIN
BEGIN_
BIT
BREAKOUT
B2500
B3500
B4700
B500
B9350
B9352

CASE
CASE_
CDATE
CHANNEL
CLOSE
COMMENT
COMMON
COMPARE
CONTROL
COPY
CRUNCH

DACCUM
DATACOMM
DATE
DCT2000
DEFINE
DEMAND
DESCRIPTOR
DISARM
DISC
DISCONNECT
DISK
DISKORPACK
DISKPACK
DISPLAY
DISPLAYUNIT
DIV
DO
DO_
DOUBLE
DOZE
DUMP

EDIT
ELSE
END
ENTER
ENTRY
EOR
EQL
ESAC
EXIT
EXITBLOCK
EXITCASE
EXITCOND
EXITLOOP
EXITROUTINE
EXTERNAL

FALSE
FI
FILE
FILL
FIND
FINDPACK
FIXED
FLOW
FORWARD
FRIDEN7311

GEQ
GET
GLOBAL

GO
GTR

HALT

IA
IACCUM
IBM1030
IBM1050
IBM1070
IF
IF_
IFF
IN
INDIRECT
INITIATE
INTEGER
INTERROGATE
IO
IX1
IX2
IX3

JDATE

LABEL
LEQ
LINES
LINKAGE
LOCK
LSS

MEMORY
MIX
MIXCALLER
MIXID
MIXNUM
MIXTBL
MOD
MUL

NEQ
NM
NO
NOT
NUMERIC

OCS
OD
OF
OFFER

OI
OLBANKING
OPEN
OR
OUT
OVERFLOW
OVERLAY
OWN

PAGE
PICTURE
POLLQ
POPQ
PORT
PORT_CURREC
PORT_DATE
PORT_EOF
PORT_ERROR
PORT_KEY
PORT_MAXMSG
PORT_Q
PORT_STATE
PORT_SUBQ
PORT_TIME
PRINTER
PROCEDURE
PROG_ENTRY
PTPUNCH
PTREADER
PULLQ
PUNCH
PURGE
PUSH
PUTQ

QUICKTIME

RACCUM
READ
READER
REAL
REEL
REINSTATE
RELEASE
REMAINDER
REMOVE
RESULTS
REVERSE
REWIND
ROUTINETYPE

SCAN
SEARCH

SEEK
SEGDICT
SEGMENT
SEGMENTED
SET
SIGNED
SINGLE
SN
SORT
SORTER
SORTER4
SPACE
SPOMESSAGE
STOP
STORE
SUBROUTINE

TAPE
TAPEGCR
TAPEPE
TAPE7
TAPE9
TC500
TC700
THEN
TIME
TIMER
TIME60
TO
TOPLOOP
TOUCHTONE
TRACE
TRANSLATE
TRUE
TT28
TWX

UA
UN
UNLOCK
UNSEGMENTED
UNTIL

VALUE

WAIT
WHILE
WITH
WRITE

ZIP

Reserved Words - Class 2

ABORT
AFTER
ALL
ANY
AREA
ASCENDING
ASCII
ASSIGN
AWAITING_HOST

BACKUP
BINARY
BLOCKED
BLOCKSTRUCTURE
BRANCH
BREAK
BUFFERS
BUR
BY

CANCEL
CENSUS
CHANGEDSUBFILE
CHANGEEVENT
CHECK
CHR
CLOSED
CLOSE_PENDING
COMPRESSION
COMPRESSIONPOSSIBLE
COMPUTER
CONDCANCEL
CRCRINPUT
CRCROUTPUT
CURRENTRECORD
CYLINDER

DATA_LOST
DCP
DEACTIVATED
DEACTIVATION_PENDING
DELETEETX
DELINK
DESCENDING
DGT
DIAL
DICTIONARY
DIALOGPROTOCOLLEVEL
DISCONNECT
DISCONNECTED
DYNAMIC

EBCDIC
ENABLE
EOP
EQUAL
EXTENDED

FILESIZE
FILESTATE
FLIPFLAG
FOR
FORMS
FPT
FROM

GIVING
GREATER
GUARDED

HOSTNAME

IGNORE
INCREMENT
INPUT
INPUTEVENT
INST
INTNAME
IOERROR

JSL
JSR

KEY

LASTSUBFILE
LINK
LOW
LOWEST

MAXCENSUS
MAXRECSIZE
MAXSUBFILES
MCP
MODE
MULTIFILE
MVC
MYHOSTNAME
MYNAME

NO_BUFFER
NO_ERROR
NO_FILE_FOUND
NONE
NOTIMEOUT

ODDPAR
ODTINPUTPRESENT
OFF
OFFERED
OFFSET
ON
OP
OPENED
OPTIONAL
OUTPUT
OUTPUTEVENT

PARITY
POLL
PRESETSTX
PRIVATE
PROCESSOR
PUBLIC

RANDOM
READY
RECORD
REM
RERUN
RETRY
RETURN
ROUTINE
RTSLRESET
RTSLSET
RUN
RUNNING

SA
SAVE
SECURITYGUARD
SECURITYTYPE
SECURITYUSE
SERIAL
SHARED
SHUTTING_DOWN
STACK
STALEMATE
STOQINPUT
STOQOUTPUT
STREAM
SUBFILEERROR
SUPPRESS

TGL
THRU

TITLE
TONE
TRANSTBL

UL
UNEQUAL
UNREACHABLE_HOST
USA
UNSUPPORTED_FUNCTION
USING

VARIABLE
VOICE

WAIT
WDS
WORK
WORKAREA
WRITEREAD
WRITEREADTRANS
WRITETRANSREAD

XCH

YOURNAME
YOURUSERCODE

ZONE

APPENDIX B
HOW TO WRITE A BPL PROGRAM

GENERAL

The writing of a computer program presupposes an understanding of the problem to be solved and the selection of a suitable programming language which will provide the most efficient solution to that problem. Assuming that these conditions are satisfied, the following considerations are presented as a guide in the writing of a BPL source language program.

WRITING RULES

The BPL compiler accepts a card image input file where columns 1 through 72 may be used for statements, declarations, or comments and columns 73 through 80 are the record sequence-numbers.

The coding may be specified in a completely free form; that is, any number of statements, declarations, or comments may appear on a single record[s] or over as many records as desired. Column 72 is considered adjacent to column 1 of the next record. Extra spaces may be used freely throughout the BPL code to improve the readability of the text.

For example, the IF statement may be written as:

```
IF X EQL Y    THEN X := 0
              ELSE X := 1
```

Where each line on the page represents a separate record.

FORM OF A BPL PROGRAM

Programs are divided into logical units called PROCEDUREs and blocks, each beginning with a BEGIN statement and terminating with an END statement. PROCEDUREs have an internal structure as described in the Declaration Statements of this manual. A PROCEDURE has a definite ordered relationship to all other PROCEDUREs within a program: either side-by-side (parallel PROCEDURE) or subordinate (nested PROCEDURE). The ordering defines the scope or range of a data-name, and the PROCEDURE(s) which may be invoked from a given PROCEDURE.

In the description that follows, the main program is considered to be the outermost block (level zero). The PROCEDURE(s) contained within the program are considered as being nested at least one level down; that is, they are on level 01 or greater. Data-names and nested PROCEDUREs which are used within a PROCEDURE must be declared before any executable statements in that PROCEDURE.

Table B-1 shows the structure of a typical, though arbitrary, BPL program. Each bracket represents a PROCEDURE and is labeled as being PROCEDURE-n (P_n) through END-n (E_n). The declarations and executable statements are indicated by D_n and X_n, where n denotes the PROCEDURE or block to which the statement belongs. Although the number and nesting of PROCEDUREs will vary among programs, the relationship of the parts, declarations, nested-PROCEDUREs, and their executable statements *must be as shown.* That is, *all* DECLAREs for a given PROCEDURE must appear in that PROCEDURE before declaring any nested-PROCEDURE(s) and before execution of any statements. However, when a nested-PROCEDURE(s) is declared, it must be completed in its entirety (including the executable statements and END;) *before* the first executable statement of the parent-PROCEDURE can be specified.

Five PROCEDUREs, three of which are on level 1 (P_1, P_2, and P_3) and two on level 2 (P_4 and P_5) are shown in figure B-1. The outer block is called the program and has a BEGIN and END statement.

Execution of an object BPL program starts at the first executable statement in the outermost block (statement X0) and is the statement which immediately follows *all nested PROCEDUREs.* Execution of statements then continues successsively from statement to statement within the outermost block until a STOP or the final END statement is encountered, which brings the program to a normal end-of-job.

Since the source code line format in BPL is very flexible, it is suggested that statement levels be indented on new lines to improve the readability of a program. Thus, each new PROCEDURE may be indented to a new margin, and its corresponding END may be placed on that same margin. Also, since statements may contain other statements (such as DO, IF, and CASE), each lower statement-level may be indented and when a higher-level is resumed, its statements should be placed at the proper level margin. It should be noted that the above is *only* a suggestion, and that identing of statements will in no way affect the operation of a BPL program.

Table B-1. A Typical BPL Program

Statements	Comments
BEGIN	
D_0	Declare global data-names (level- 0).
D_0	
P_1	Begin PROCEDURE 1.
D_1	PROCEDURE 1's local data declarations.
D_1	
X_1	PROCEDURE 1's executable statements.
X_1	
E_1	END of PROCEDURE 1.

Table B-1. A Typical BPL Program (Continued)

Statements	Comments
P_2	PROCEDURE 2 (level-2). Local data.
D_2	
D_2	
.	
.	
.	
X_2	PROCEDURE 2's executable statements.
X_2	
.	
.	
.	
E_2	END of PROCEDURE 2.
P_3	
D_3	PROCEDURE 3's local data-names that are also global to PROCEDUREs 4 and 5 (level-1).
D_3	
P_4	PROCEDURE 4's local data-names (level-2).
D_4	
D_4	
.	
.	
.	
X_4	PROCEDURE 4's executable statements.
X_4	
E_4	END of PROCEDURE 4

Table B-1. A Typical BPL Program (Continued)

Statements			Comments
		P_5	PROCEDURE 5's local data-names (level-2).
		D_5	
		.	
		.	
		.	
		X_5	PROCEDURE 5's executable statements.
		E_5	END of PROCEDURE 5.
	X_3		PROCEDURE 3's executable statements.
	X_3		
	.		
	.		
	.		
E_3			END of PROCEDURE 3.
X_0	——		First executable statements in program.
X_0	——		Last executable statements in program.
X_0			

END;

A study of the examples given, with the detailed descriptions of the BPL statements and declarations as provided elsewhere in this manual, should aid in a better understanding of how a BPL Program is written.

PROCEDURE CALLING

Any PROCEDURE may call (invoke) any other PROCEDURE which is currently invoked (any direct-ancestor) or any PROCEDURE which is nested one level down within a currently invoked PROCEDURE (any first-generation descendent).

For definitional purposes, the program is considered as being the outer-most PROCEDURE and is always in a currently-invoked status.

The rule follows directly from the concept of scope. Each PROCEDURE passes all of its declared names, as globals, to all of its descendents. This includes the names of all PROCEDUREs nested one level down. Notice the difference between the name of a PROCEDURE on the current level and the PROCEDURE being named which is on the next lower level.

Relationships

Let figure B-1 depict the compile time relationships of the specified PROCEDUREs.

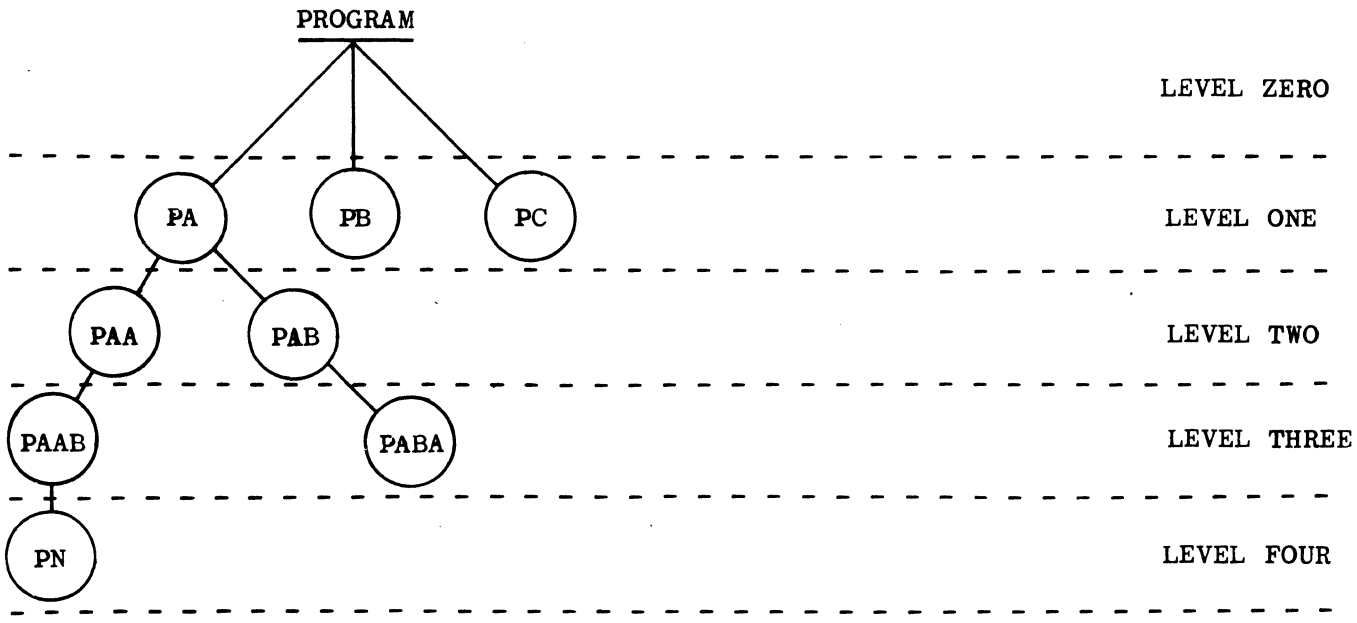


Figure B-1. Compile Time Relationships of Procedures

Then the SCOPE or range of each PROCEDURE is as follows:

- PROCEDURE *PN* may invoke any of the following: PN, PAAB, PAA, PA, PAB, PB, or PC.
- PROCEDURE *PB* may invoke any of the following: PA, PB, and PC.
- The parent-PROCEDURE may invoke PA, PB, PC.
- PROCEDURE *PAB* may invoke PAB, PABA, PA, PAA, PB, and PC.

As another example, let A, B, C, D, L, M, and K be the names of a set of PROCEDUREs. If the compile time relationship of the PROCEDUREs is:

A (B (K), C (L, M), D)

Then the SCOPE of a PROCEDURE-invoking statement in each PROCEDURE is:

- *A* may call A, B, C, or D.
- *B* may call B, K, A, C, or D.
- *K* may call K, B, A, C, or D.
- *L* may call L, C, M, A, B, or D.
- *M* may call M, C, L, A, B, or D.
- *D* may call D, A, B, or C.

This example could be represented as shown in figure B-2.

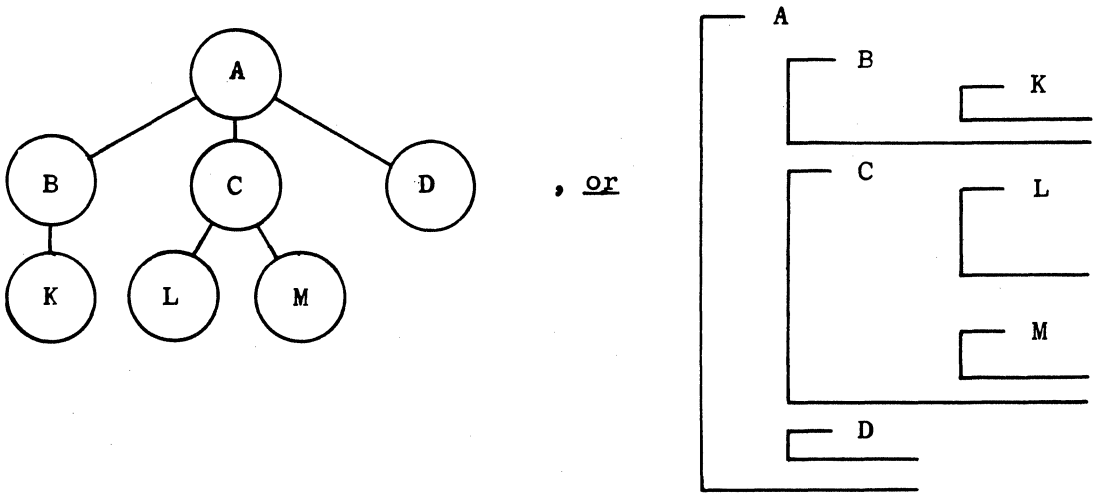


Figure B-2. The Scope of a Procedure

TABLE CREATION

Any contiguous data group can be treated as a contiguous record or table. The following is an example of a table named INPUT_AREA:

```
ALPHA INPUT_AREA (80);
ADDRESS = INPUT_AREA;
ALPHA LAST_NAME (15);
     ALPHA MID_INITIAL (1);
     ALPHA FIRST_NAME (16);
     INTEGER  SS_NO (9);
     INTEGER  USE_CODE (1);
ALPHA      COMMENT_AREA (43);
ADDRESS;
ALPHA      END_TABLE (0);
```

This same table could have been written without the first, second, and ninth lines and referred to as LAST_NAME THRU END_TABLE.

```
&SAMPLE PROGRAM*********************************************

  SAMPLE:                                              & OPTIONAL PROGRAM ID.

  &
  &             ********************************************
  &             THIS IS A SAMPLE BPL PROGRAM TO PERFORM
  &             A CARD TO TAPE OPERATION.THE NUMBER OF
  &             CARDS READ IS DISPLAYED UPON THE ODT.
  &             ********************************************
  &
          BEGIN                                        & ONE BEGIN IS REQUIRED
                                                       & AT THE BEGINNING OF
                                                       & EACH PROGRAM.
  LABEL EOF, LOOP;                                     & LABEL DECLARATIONS
                                                       & MUST APPEAR BEFORE
                                                       & INSTRUCTION CODING.
  INTEGER COUNTER (6):= 0;                             & COUNTER PRESET TO 0.
  ALPHA DISPLAYEDCOUNTER (6);                          & COUNTER FOR DISPLAY.
  ALPHA WORKSPACE (80) MOD 4;                          & COMMON WORK AREA
  FILE CARDIN, READER,
      RECORD WORKSPACE 80, BUFFERS 2;                  & CARD INPUT
  FILE TAPEOUT, TAPE,
      RECORD WORKSPACE 80, BUFFERS 2;                  & TAPE OUTPUT
          OPEN IN CARDIN;                              & OPEN CARD INPUT FILE.
          OPEN OUT TAPEOUT;                            & OPEN TAPE OUTPUT FILE
  LOOP:                                                & PREVIOUSLY DECLARED
                                                       & LABEL.
          READ CARDIN [EOF];                           & READ CARD
          COUNTER := COUNTER + 1;                      & INCREMENT COUNTER
                                                       & BY 1.
          WRITE TAPEOUT;                               & WRITE CARD TO TAPE
          GO TO LOOP;                                  & RETURN TO READ.
  EOF:                                                 & WHEN LABEL IS USED IT
                                                       & IS FOLLOWED BY A
                                                       & COLON (:).
                                                       & INSTRUCTIONS ARE
                                                       & TERMINATED WITH A
                                                       & SEMICOLON (;).
          CLOSE CARDIN;                                & CLOSE INPUT CARD
                                                       & FILE
          CLOSE TAPEOUT LOCK;                          & CLOSE OUTPUT TAPE
                                                       & FILE WITH LOCK.
          DISPLAYEDCOUNTER := COUNTER;                 & CONVERT TO ALPHA MODE
          DISPLAY DISPLAYEDCOUNTER;                    & TYPE COUNT ON SPO
                                                       & ONE END IS REQUIRED
                                                       & FOR EACH BEGIN.
                                                       & ALSO THE LAST END
                                                       & WILL GENERATE A
  END:                                                 & STOP RUN COMMUNICATE.
```

```
  SAMPLE:                                              & Optional program ID
&  &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&  This sample BPL program lists and reformats a diskpack file.
&  The number of records read is displayed at the ODT.
&  A procedure is used, to illustrate BPL program structure.
&  &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

BEGIN                                          & BEGIN is required at outermost block

LABEL EOF;                                     & Labels used within each block must be
                                               & declared before that block's
                                               & statements
INTEGER COUNTER (06) := 0;                     & Counter preset to zero
BIT     MORE_DATA    := TRUE;                  & Boolean flag
ALPHA   INREC   (80) MOD 4,                    & Must declare file record areas before
      OUTREC (132) MOD 4;                      & FILE, and must be MOD 4.
ADDRESS = INREC;                               & Redefine
 ALPHA  IN_NAME (40),                          & fields
      IN_ADDR (40);                            &  within INREC
ADDRESS;                                       & End redefinition

FILE INFILE,                                   & Input file (name defaults to "INFILE")
   DISKPACK 20 BY 500,                         & Pack, 20 areas, 500 records/area
   RECORD INREC 80,                            & Record area is INREC.
   BUFFERS 2, BLOCKED 9;                       & 2 buffers, 9 records per block
FILE OUTFIL,                                   & Outut file
   PRINTER, "SAMPLE",                          & Creates a print file named "SAMPLE"
RECORD OUTREC 132;                             & Record area is OUTREC.

   &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
   & Procedures must be declared before the executable instructions
   & in the block
   &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

PROCEDURE DISPLAY_RESULTS;
   BEGIN
        OWN ALPHA   DISPLAY_DATA (21) := & To display count; OWN is
             "Records read = xxxxxx"; & required to
                                          preinitialize
          DISPLAY_DATA.+15.6.NM := COUNTER; & Convert to display mode
          DISPLAY DISPLAY_DATA;             & Display it
     END;
```

```
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
& First executable statement is next
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

      OPEN IN INFILE;                          & Open input file
      OPEN OUT OUTFIL;                         & Open output file
    WHILE MORE_DATA  DO_                       & Begins WHILE...DO_ loop
        READ INFILE [EOF];                     & Read record
        COUNTER          :=COUNTER + 1;        & Increment counter
        OUTREC           :=IN_NAME;            & NAME field to output record
        OUTREC.+50.40 := IN_ADDR;              & Put ADDR 10 bytes past name
        WRITE OUTFIL SINGLE;                   & Write output, single-spaced
        TOPLOOP;                               & Return to beginning of loop
    EOF:                                       & End_of_file label from READ
        MORE_DATA :=FALSE;                     & Reset boolean flag
    OD;                                        & End of WHILE...DO_loop

    CLOSE INFILE;                              & Close files
    CLOSE OUTFIL;                              &
    DISPLAY_RESULTS;                           & Procedure call
END;                                           & End program
```

APPENDIX C
WARNING AND ERROR MESSAGES

GENERAL

The following is a list of warning and error numbers with their respective descriptions.

Warning numbers are four digit numbers where the first two digits range from 00 through 09 and the last two digits represent variations within each warning category. Currently the warning categories are:

00	Sequence error (warning only).
01	Receiving field warning.
02	Limit warning.
03	Address warning.
04	Controller warning.
05	Declaration warning.
06	Through 09 are not used.

Error numbers are four digit numbers where the first two digits range from 10 through 99 and the last two digits represent variations within each error category. Currently the error categories are:

10	Excess operands.
11	Subscript error.
12	Controller error.
13	Illegal operand.
14	Missing special character.
15	Duplicate word.
16	Illegal word.
17	Unidentified word.
18	Missing reserved word.

19 Illegal literal.

20 Illegal statement.

21 Invalid character.

22 Begin/and error.

23 Illegal declaration.

24 Duplicate declaration.

25 Incomplete statement.

26 Illegal combination.

27 Limit exceeded.

28 Missing key word.

29 Compiler error.

30 Illegal special character.

31 Illegal picture declaration.

32 Expression errors.

33 Through 99 are not used.

NOTE

All error and warning messages printed by BPL contain a line of XXXX---X to indicate the column in error. The number of "X"s indicate the column in error. The error is always printed after the symbolic line in error. It is not necessarily printed immediately after the error condition due to scanning considerations. Most errors will be in the two symbolic records immediately preceding the error message.

WARNINGS

00 Sequence errors.

0000	Sequence error.

01 Receiving Field Warnings

0100	Receiving field too large for logical operation.
0101	Dividend too large to move to REMAINDER. Possible run time error condition.
0102	Receiving field too large to contain MEMORY.
0103	Sizes are not equal when using [WDS] override. The smaller size is used.
0104	The sending and receiving fields for TRANSLATE are not equal. The smaller size is used.
0105	Receiving field too small for literal. The literal is truncated.
0106	Receiving field for subscript computation too small if subscript has maximum possible value.
0107	Indirect field length for receiving field on ADD, SUBTRACT, or MULTIPLY. Possible run time overflow.
0108	Data name less than 6 digits. 6 digits will be used for timer instructions.
0109	Receiving field smaller than maximum attribute size. Possible truncation.
0110	Receiving field size should be less than the JOBINFO response area size.

02 Limit Warnings

0200 Compiler could not assign as mush dynamic space as was requested. Summary will indicate what amount of memory space was actually obtained.

0201 Requested core too small. Assigned core is shown in the summary.

0202 Not enough memory for SORT intrinsic.

0203 The REAL number used in this statement has been truncated.

0204 Coding exceeds 300KD. Any BCT's or ENTER's above 300KD will given specified results.

0205 Data name greater then 30 characters. Truncated to 30.

0206 Eight bit data names are limited to 150 characters if initialized.

0207 Alpha literal too long. Truncated to maximum size for specified attribute.

0208 Identifier size exceeds maximum attribute size. Possible truncation.

03 Address Warnings

0300 A stack relative data name sent as a name parameter will generate an invalid address if the stack is at an address over 100KD.

0301 The DIVIDE instruction used indirect length fields on the dividend and quotient/ divisor such that REMAINDER length cannot be determined. The actual dividend will be used to store the REMAINDER.

0302 An increment/decrement controller has been used with a name parameter. Resultant address will be offset to some stack address by the increment/decrement.

0303 An address constant, alpha constant or instruction would have occurred on an odd address. One digit of fill was added.

0304 Indirect data names used in arithmetic operations with CONTROL EXTENDED set have their addresses incremented by two digits and their size decreased by two digits.

04 Controller Warnings

0400 An index register has been used as a controller override on an operand having implied index usage, i.e., buffer or stack relative.

0401 When indirect field length is used, the compiler cannot determine field sizes in an arithmetic expression.

0402 A UN result is assigned to a receiving field with a UA or an SN controller.

05 Declaration Warnings

0500 CONTROL OP B4700 is not set, so the following FIXED declarations are processed as indicated:

```
FIXED INTEGER A; - SIGNED INTEGER A(7);

FIXED REAL B; - REAL B (8);

FIXED DOUBLE C; - REAL C (16);
```

0501 Duplicate setting of CONTROL OP declaration: OP 4700 or OP 3500 was previously set when CONTROL statement was encountered.

0502 Global greater than 6 characters. Characters used as 2 digit hexadecimal.

0503 Not enough room in dictionary for entry. If not accompanied by 2932, when an extra forward declaration name of the same length was found.

0504 Multiple buffers disallowed. Defaulted automatically to 1.

0505 File must be unblocked. Defaulted automatically to 1.

0506 Segmentation directions disallowed for Type II and Type III ICMs.

0507 You may produce unpredictable results if you use both options of the mix function syntax in the same program.

0508 The DATACOM ACCEPT and DATACOM DISPLAY constructs are not implemented on operating system MCP/VS 2.0 and greater.

ERRORS

10 Excess Operands

1002 Excess operands in a DATACOMM statement.

11 Subscript Errors

1100 Literal subscript caused resultant address to be negative

1101 Subscript variable is not an index register, data name or literal.

1102 The subscript variable used within an "ADDRESS OF" construct contains other than a numeric literal.

1103 Subscript not numeric.

1104 Subscripting is being attempted on a data name already indexed, i.e., double indexing.

1105 FIND, MIXID, and MIXNUM cannot have an array element as program-id or file-name.

12 Controller Errors

1200 One or more of the following basic controller errors have been found by the scanner:

1. Double setting of IA.
2. Double setting of hardware usage options, i.e., UN, UA, SN.
3. Illegal word as a controller variable.
4. Double setting of index registers.
5. Double increment/decrement.
6. Numeric literal 6 digits.
7. Double length overrides or zero length.
8. Double indirect field length labels.
9. Illegal use of a length with an indirect field length.
10. A function output parameter must be the only override.

1201 A controller other than an index register is associated with the REAL-DOUBLE operand.

1202 An undigit cannot be used as an override.

1203 The indirect field name used as a controller variable has a base relative address 38, 36 if SRD.

1204 The indirect field name has a base relative address below 40 but is an uneven address.

1205 The decrement used as a controller override causes the address to go negative for the segment.

1206 SN controller not allowed in scans; UN not allowed on SCAN ZONE.

1207 An indirect field controller has been used illegally in association with this bit type statement or logical type statement.

1208 Controller fields of usage INFL, LENGTH, or IA may not be used in conjunction with a bit declaration.

1209 An indirect field length cannot be used with the "beginning of search" address in a SEARCH statement.

1210 A label has an increment or decrement which is greater than + 9999.

1211 An indirect field length has been used with a bit, label, or procedure name.

1212 Function output overrides are only allowed on port files.

1213 INFL not allowed when setting an attribute.

1214 An IA, UA, or NM was used on a field that does not start at a MOD 2 location.

1215 Controllers are not allowed with any of the following reserved words: DATA, JDATE, TIME, TIME60, MEMORY, FIND, FINDPACK, MIX, MIXCALLER, MIXID, MIXNUM, MIXTBL, JOBINFO.

1217 IA not allowed on first list entry in SEARCH LINK/DELINK.

1218 SN not allowed as a search key controller in SEARCH LINK/DELINK.

1219 A when indirectly addressed and used in a logical comparison must have a length of one.

1220 Controllers are not allowed on the receiving field for MIXTBL.

1221 Controllers are not allowed with an [EOF] label.

1222 INFL field used as A_FIELD for SRD must be of length 4.

13 Illegal Operands.

1300 A literal cannot be a receiving field.

1301 AF or BF field size exceeds 100.

1302 "REMAINDER" has been used with no prior DIVIDEs in this segment.

1303 Parameter block address must be MOD 4.

1305 One or more of the operands is signed numeric in this Boolean expression.

1306 Receiving field too small.

1307 The data name in DOZE is greater than 5 digits long.

1308 In a ZIP statement the data name must be ALPHA with no IA or indexing used.

1309 REAL arithmetic requires signed operands.

1310 A TRACE statement does not contain a literal, or contains a literal that is not UN, that is greater than 2 digits long or that has a value greater than 3.

1311 The increment in the SEARCH statement is not a literal or an indirect field address.

1312 Illegal receiving field in assignment.

1313 The operand in this arithmetic statement is ALPHA.

1314 SPOMESSAGE receiving field size is less than 160 digits.

1315 An accumulator operand has been used but the CONTROL statement has not indicated B4700 OP codes.

1316 The increment cannot be calculated in the SEARCH construct when INCREMENT ADDRESS is used because of one of the following:

1. A data name does not follow the word ADDRESS.
2. The data name used after ADDRESS is on a different base then the search field name.
3. Indexing with different index registers on the two fields.
4. Indirect addressing is used on either of the two fields.
5. Either one of the two fields is a label.

1317 A data name used with a fixed length operand must be at a MOD 4 address and correct length for INTEGER, REAL, or DOUBLE.

1318 In SPOMESSAGE, the sending field must be a data name or literal and the receiving field must be a data name.

1319 A data name is required for INTERROGATE ADDRESS that is INTEGER (6), INTEGER (16) for INTERROGATE.

1320 A FIB name is required as the first entry in a DATACOMM statement.

1321 A WAIT variable must be declared INTEGER (5).

1322 The sending or receiving field for ACCEPT or DISPLAY from remote SPO must be ALPHA and less than 100 characters.

1323 The ACCEPT or DISPLAY SPO-id must be an ALPHA data- name, less than 60 characters.

1324 The TRANSLATE table must be at a MOD 1000 address. A literal address is not allowed.

1325 "ADDRESS OF" cannot be used as an operand in arithmetics as the @C@ will cause erroneous results.

1326 All fields in a move data instruction must be at MOD 4 addresses.

1327 Missing file name or program name. Must be ALPHA literal of 6 characters or less.

1328 OVERLAY requires segment number less than 999.

1329 The response area for DATACOMM EXTENDED must be a minimum of 26 digits in size.

1330 A mantissa size of zero has been generated. A signed field of size 3 or less has been used as a REAL operand while the exponent requires 3 digits of the total length of this signed field to be used as a REAL operand.

1331 Sort key not within the sort record.

1332 Location for loading the mix table must be MOD 2.

1333 Sort record must be MOD 4 in length.

1334 Delay time for ACTION 8 MICR must be 4.UN.

14 Missing Special Characters

1400 Missing comma.

1401 Missing right bracket.

1402 Missing semicolon.

1403 Missing right parenthesis.

1404 Missing left bracket.

1405 Missing left parenthesis.

1406 Missing colon.

1407 Missing assign.

1408 Missing comma or right parenthesis in PROCEDURE call parameter string.

1409 Missing "=" in a DEFINE declaration.

1410 Missing assign or left parenthesis.

1411 Missing right parenthesis, assign, or semicolon.

1412 Missing left parenthesis in a PROCEDURE call declared with parameters.

1413 Missing left parenthesis for a parametric DEFINE. Even if it is called with a null argument list it must have the parenthesis.

1414 An equal sign is expected.

1415 A period is expected.

15 Duplicate Words

1500 Duplicate declaration.

1501 File be same name.

1502 Duplicate ODTINPUTPRESENT in the wait statement.

1504 Duplicate label in this segment.

16 Illegal Words

1600 An identifier must start with a letter, not a number.

1601 An identifier did not follow the reserved word PORT in the port declaration.

1602 An identifier did not follow the reserved word FILE in the FILE declaration.

1603 An illegal key word has been found in the FILE declaration.

1604 An identifier did not follow the key word KEY in the file declaration.

1605 An illegal word follows MOD in the DYNAMIC declaration. Should be a literal 2 or 4 only.

1606 Illegal word inside brackets.

1607 This declaration contains either:

1. An illegal word inside parentheses not being an integer or the word DYNAMIC.

2. A reserved word has been used as an identifier.

1608 Illegal word following MOD. Must be a numeric literal not exceeding 4 digits long and not 0. If local variable it may only be MOD 2.

1609 An identifier does not follow the reserved word PROCEDURE or SUBROUTINE.

1610 Illegal parameter specified in this PROCEDURE declaration.

1611 Illegal word as a primary declarative or statement starter.

1612 Illegal word in FILE declaration.

1613 An illegal word follows key word LABEL in FILE declaration.

1614 Illegal word in I/O statement.

1615 Missing input file name following the reserved word SORT.

1616 Missing output file name in SORT statement.

1617 Illegal word in SORT statement.

1618 Illegal CLOSE option in SORT statement.

1619 Illegal PARITY type in SORT statement.

1620 Illegal sort key in SORT statement.

1621 A data name must follow operator or assign.

1622 A reserved word has been used illegally or a unary + or - is not followed by a numeric literal.

1623 An illegal word has been found in the head portion of the CASE statement.

1624 An illegal word has been found in the TRACE, DUMP, ZIP, DOZE, FILL, STOP, EXITROUTINE, ACCEPT, DISPLAY.

1625 An illegal word has been found or a word missing in the port declaration.

1626 Illegal file attribute name.

1627 An identifier did not follow the reserved word SET in the set statement.

1628 Illegal use of override, for example, [ALL], [JSL], [XCH], and so forth.

1629 Illegal use of reserved word ELSE.

1630 A word other than a data name or label follows DUMP.

1631 An illegal word has been found in declarations.

1632 TRUE and FALSE used in context illegally.

1633 Illegal word found in parameter list when PROCEDURE called, this is, mismatched parameters.

1634 Illegal word as statement starter.

1635 Illegal buffer area in FILL statement.

1636 Illegal program-id in FILL statement.

1637 Illegal action label in FILL statement.

1638 Illegal formal parameter in a parametric DEFINE. Should be a BPL identifier.

1639 Illegal word in EDIT statement; Should be a data name, label, or picture

1640 In-line constants using the STORE statement can only be initialized to literal values.

1641 Illegal word as entry point; must be a data name, label, or procedure.

1642 Illegal word as EXIT point; must be a data name, label, or procedure.

1643 Illegal word used in a TRANSLATE statement. All fields must be data names or labels.

1644 An illegal word has been used as an arithmetic operand.

1645 Illegal word in move data; must be a data name, label, or procedure.

1646 Entity following TRANSLATE in SORT statement is not an alpha literal.

1647 A branch type instruction must usa a data name, label or procedure name.

1648 A stack relative variable may only be referenced within the level of nesting in which it was declared.

1649 Illegal word in MOVE LINKS; must be a label, data name, or procedure.

1650 EXITCOND, EXITCASE, EXITLOOP, TOPLOOP does not appear within the appropriate type statement.

1651 ODDPAR valid with type file only.

1652 An illegal word has been found or a word missing in the get statement.

1653 An illegal word has been found or a word is missing in the get statement.

1654 Illegal option for the wait statement.

1655 Record address was declared as fixed. Cannot be changed.

1656 One of the following problems has been found.

1. INFL not allowed
2. Wrong size.
3. Wrong type.
4. Not a data name.

1657 File must be a port file.

1658 The specified attribute cannot be set.

1659 Identifier is too small to receive results information.

1660 Unrecognized or illegal word found in Privileged Instructions.

1661 Module name is required for Type II ICMs.

1662 Illegal word in the wait statement.

1663 An identifier did not follow the reserved word GET in the get statement.

1664 Missing filename in the wait statement.

1665 Missing identifier in the wait statement.

17 Unidentified Words

1701 Word not found in dictionary.

1702 Parameter declaration not found in formal parameter list.

1703 LABEL was declared, used, but did not appear.

1704 Missing module or program name.

18 Missing Reserved Words

1800 Missing reserved word INTEGER after SIGNED.

1801 Missing reserved word PROCEDURE after SEGMENTED.

1802 Missing reserved word THEN.

1803 Missing delimiter.

1804 Missing reserved word DO.

1805 Missing reserved word UNTIL.

1806 Missing reserved word [XCH]. Received 3 operands - believed to be a 3 way move link.

1807 Missing reserved word IN or OUT in FILL statement.

1808 Missing reserved word following FILL in this DATACOMM construct. Should be READ, WRITE, WRITEREAD, WRITEREADTRANS.

1809 Missing "ADDRESS;" required to unstack "ADDRESS=".

1810 Missing reserved word MODE.

1811 Missing keyword "ON" before family name.

19 ILLEGAL LITERALS

1900 Special literal within % or @is not range 0 through 9, A through F.

1901 Special alpha literal (delimited by %) is not an even number of digits.

1902 Literal exceeds 99 digits or 99 characters. Truncated to 99.

1903	Number of areas (rows) in the FILE declaration exceeds maximum allowed by the MCP, namely 100.
1904	Number of records per area in the FILE declaration exceeds 8 digits in length.
1905	RECORD size is either not a numeric literal or exceeds 5 digits in length in the FILE declaration.
1906	A FIXED INTEGER literal may not exceed 7 digits in length.
1907	Literal exceeds 6 digits in length.
1908	The SAVE FACTOR literal in the FILE declaration exceeds 3 digits in length.
1909	Illegal literal following RERUN in the FILE declaration.
1910	Illegal associated with BUR standard label in the FILE declaration is not an alpha literal, or exceeds 44 characters.
1911	Illegal literal following BUFFERS in the FILE declaration.
1912	Illegal literal following BLOCKED in the FILE declaration.
1913	Illegal literal following TRANSLATE in the FILE declaration.
1914	Illegal channel number in WRITE statement. Literal exceeds 2 digits.
1915	Literal must be 1 or 2 digits long in Boolean conditional.
1916	Literal greater than declared field length.
1917	Literal mask in bit test cannot be signed.
1918	Literal mask in bit test cannot be Boolean.
1919	A logical expression contains an illegal signed literal.
1920	A LIBRARY file name is either not an alpha literal or exceeds 6 characters in length.
1921	A literal is required after the decimal point of a REAL number.
1922	An "ADDRESS OF" literal is not allowed by B4700 fixed length arithmetics.
1923	Only integer literals can be used in the SPACE construct.
1924	For a print file, a literal for channel or lines cannot be more than 2 digits. For any other file, it cannot be more than 4 digits.
1925	Record size is not a numeric literal or exceeds six digits in length in the port declaration.
1926	Length for in-line STORE must be an integer literal less than or equal to 6 digits.

1927 Literal larger than size given.

1928 A literal used as the character count for an ENTER statement must not exceed 4 digits.

1929 A literal sent as a parameter contains more significant digits than the formal declaration.

1930 Literal in "ASSIGN BY nn" must be 99 or less.

1931 Literal exceeds 1 digit in length.

1932 Literal length must be MOD 2.

1933 Literal not allowed.

1934 Literal greater then:

4 for SRD, HBK, and SMF;

6 for STT.

1935 Variable size exceeds 4 digits in length in file declarations.

1936 Formal declaration of parameter too large to generate literal.

1937 Support index is not a numeric literal or exceeds four digits in length in the port declaration.

1938 Illegal value for a mnemonic attribute.

1939 Literal type does not match attribute type.

1940 Numeric literal exceeds maximum length for specified attribute.

1941 Illegal attribute value.

1942 Literal exceeds maximum length or value exceeds 86400 in the wait statement.

1943 Literal exceeds six digits in length.

20 Illegal Statements

2001 Illegal entity following GO.

2002 Illegal use of THEN.

2005 An arithmetic statement contains an invalid arithmetic operator.

2006 More than one move required when [TGL], IA, or IFL was specified.

2007 An override of [WDS], [CHR], or [XCH] specified and one of the following conditions occured:

literal sending field.

addresses not MOD 4 or MOD 2.

length not MOD 2 char if [WDS].

if [XCH], sizes unequal, greater than 100, or usage are not equal.

if [WDS], operands must be of equal word size.

NOTES

1. Appropriate syntax checks are inhibited when IA, IFL or indexing is indicated.
2. On [WDS], if IFL specified for one, must be specified for both. If [JSL] or [JSR] and one of the following conditions occurred:
   - IFL is specified.
   - IA is specified and move cannot be accomplished in one instruction.
   - The size of either operand is greater than 100.
   - The receiving field is alpha; the sending field is signed integer.

2008 On a numeric move when one or both operands exceed 100 and the receiving field is signed, both must be the same MOD 100 (i.e., 101-200, 201-300, etc.).

2009 [ALL] option literal or data name sending field may not be signed or exceed 100. [XCH] does not allow a literal.

2011 Illegal operand in bit test.

2012 The unsigned numeric data name is being compared to 0 using a LSS or GEQ relation, or to @F@using GTR or LEQ relation.

2014 Literal used following IF not allowed - two literals illegal in bit test.

2015 Length too long for comparison.

2016 Missing CASE statement variable or missing NO.

2017 Size of the CASE statement variable exceeds 6 digits in length.

2018 If the first operand is IA in a SEARCH statement the THRU option must be used.

2021 Left bracket has appeared; an "ADDRESS OF" construct is assumed which is not acceptable.

2023 RECORD key word and record name omitted.

2024 A PROCEDURE has been called from within a "PROCEDURE." construct.

2028 Illegal MICR IO construct.

2031 Illegal conditional compile statement.

2032 Expression is not followed by a valid delimiter.

2033 Either a nested COPY statement or an attempt has been made to COPY while creating a LIBRARY file.

2034 The first list entry in SEARCH LINK/DELINK must be a MOD 2 address.

2035 The entity following parenthesized argument string in a parametric DEFINE is not a "," or a ")".

2036 An invalid word follows DATACOMM.

2037 An illegal entity follows @LIBR or @ICM.

2038 More data is only valid when CONTROL OP B4700 is declared.

2039 "NO INPUT" and "NO OUTPUT" may not both be declared in a DATACOMM TRANSTBL statement.

2040 DATACOM ACCEPT and DATACOM DISPLAY constructs are not supported under MCP/VS 2.0.

21 Invalid Characters

2100 Invalid character found in SCAN.

22 Begin/End Errors

2201 Program out of bounds on END that matched with first BEGIN, that is, information appears after final END.

2202 Missing first BEGIN after program name.

2203 Missing BEGIN within PROCEDURE.

2204 BEGIN/END error in CASE statement.

2205 Illegal presence of reserved word BEGIN.

2207 Too few ENDs found on end of program ("END;" generated internally for other passes).

2208 No BEGIN after COMMON.

2209 Missing ender type FI, OD, or ESAC.

23 Illegal Declarations

2300 Right hand portion of equivalence is illegal.

2301 Invalid "ADDRESS =" declaration.

2302 The identifier's declared size exceeds 999999.

2303 Word following VALUE is not an identifier.

2304 Illegal DYNAMIC declaration; either a numeric literal does not follow ">" or the numeric literal exceeds 6 digits in length.

2305 Illegal CONTROL declaration.

2306 The DEFINE declaration does not have an identifier following reserved word DEFINE.

2307 Illegal ARRAY declaration; no ARRAY name, number of entries not a literal, or BIT is used with ARRAY.

2308 The identifier declaration has an illegal preset format; i.e., the entities following the replacement sign (left arrow or :=) are in error.

2309 Illegal LABEL declaration.

2310 Illegal declaration within a PROCEDURE declaration.

2311 The declaration within the actual PROCEDURE differs from the declaration in the FORWARD PROCEDURE, i.e., formal parameter declarations differ.

2312 FIXED INTEGER, FIXED REAL, or FIXED DOUBLE has been declared as a local variable (stack relative) in a PROCEDURE. Cannot guarantee MOD 4 in stack unless $STK4 used.

2313 Global labels not permitted in Type II ICMs.

2314 A declaration cannot be pre-initialized in DYNAMIC space.

2315 A data name declaration is being preset using the [ALL] option and is not an even multiple of the presetting literal.

2316 The record name used within the FILE declaration is not MOD 4 in assigned address.

2317 A label does not appear in the segment it was declared in.

2319 WORKAREA or RECORD may not be declared as stack relative.

2320 The WORKAREA or RECORD length is not MOD 4 in the FILE declaration.

2321 The RANDOM KEY length is not 8 digits.

2322 Records per area missing

2323 No hardware type has been specified in the FILE declaration.

2324 No RECORD name has been specified in the FILE declaration.

2325 The reserved word OWN is followed by an illegal declaration.

2326 Cannot pre-initialize data in an absolute memory location greater than base relative 200 or start of segment dictionary, whichever is smaller.

2327 The word following MULTIFILE is not a non-numeric literal.

2328 ENTRY must be to a global procedure. Possible BEGIN-END error.

2329 An index is not allowed on port attributes.

2330 A PROCEDURE has been declared within coding for this segment. If the procedure is not in coding, there is a missing END in the preceding procedure.

2331 CONTROL EXTENDED has been declared after an INDIRECT declaration.

2332 Cannot pre-initialize data in absolute memory greater than the size of main segment (MCPP option only).

2333 An INDIRECT declaration is being preset to an "ADDRESS OF" construct with a ".NO" controller (CONTROL EXTENDED set).

2335 A DYNAMIC declaration occurred illegally while producing an Independently Compiled Module.

2336 An attribute value is required.

2337 More than 100 named COMMON blocks is illegal.

2338 Unnamed COMMON is not allowed.

2339 Illegal declaration within COMMON.

2340 Illegal LINKAGE declaration.

2341 More than 9 names in LINKAGE statement is illegal.

2342 Unnamed declaration cannot be equated.

2343 Cannot "ADDRESS =" to an absolute address within COMMON.

2344 Cannot make equivalences from one COMMON area to another.

2345 A declaration within COMMON must have an identifier.

2347 Cannot preinitialize a declaration less than 6 digits long with an "ADDRESS OF" construct.

2348 The word DEFINE was found within a DEFINE declaration. Probably missing previous "#".

2349 An identifier does not follow PICTURE declaration.

2350 A mask not provided in PICTURE declaration.

2351 The word following FIXED is not INTEGER, REAL, or DOUBLE.

2352 Illegal entity following the ASSIGN portion of a FILE declaration.

2353 A stack relative variable is being initialized.

2354 A FILE declaration with multiple buffers or blocking must contain either IX2 ON or a WORKAREA.

2356 An attempt has been made to initialize data equated to declarations in another program segment.

2357 An attempt has been made to declare a file or port with the address equated to another program segment.

2358 An ICM must be unique to the segment in which it is declared. It must have a MOM (parent) segment of 001.

2359 "ADDRESS =" has requested an address that occurs prior to the start of the segment.

2360 A data name within an ICM is being initialized to an "ADDRESS OF" greater than 9999.

2361 A declaration has appeared after coding within a PROCEDURE.

2362 SUBROUTINE may only appear within a ICM.

2363 DISK or DISKPACK files may not be declared unlabeled.

2364 Illegal backup request.

2365 Illegal type appears as first declaration in global block.

2366 More than 10 parameters declared for procedure.

2367 IX2 must be off.

2368 WORKAREA for Reader-Sorter 4A Control or DLP must be 780 digits.

2369 Global code not allowed in Type II ICMs.

2370 Missing record clause in the port declaration.

2371 An index must be specified for all subport attributes.

2372 Must close with release.

24 Duplicate Declarations

2400 Duplicate declaration in CONTROL statement.

2401 Duplicate LINKAGE declaration.

25 Incomplete Statements

2500 Missing third operand in a Boolean statement.

2503 TRACE requires an operand.

26 Illegal Combinations

2600 The MODE is an illegal combination with the file type in the FILE declaration.

2601 The total lengths of the formal and actual parameters to a PROCEDURE differ.

2602 When [REM] is used, the dividend and quotient cannot be the same field.

2603 A new work area in a READ or WRITE statement must be at a MOD 4 address.

2604 When [REM] is used, the quotient size must equal the dividend size minus the divisor size.

2605 Time option must be first.

2606 Identifier and attribute are not compatible.

2607 Cannot have two integer or two alpha identifiers in port read or write statements.

2608 A SORTER use routine is present but the file is not for a reader-sorter.

2609 OPEN is followed by FLOW or DEMAND but the file is not for a reader-sorter.

2610 A UN operand has been used with a UN operand illegally in a Boolean expression.

2611 A BIT declaration cannot appear as a named parameter to a procedure.

2612 One but not all operands have indirect field lengths in 3 way more links, i.e., [XCH].

2613 Lengths differ for 3 way more links, i.e., [XCH] or possible missing ender to previous statement.

2614 Usages differ for 3 way move links, i.e. [XCH].

2615 Cannot have a FIXED REAL or FIXED DOUBLE as a value parameter to a procedure. Cannot guarantee MOD 4.

2616 Dividend and divisor have the same length causing a hardware error when executed. Could not generate extra move as [REM] was specified.

2617 CHANNEL allowed only with a print file.

2618 REVERSE not allowed with a print file.

2619 No indirect field length allowed on table offset for SEARCH/LINK/DELINK or SEARCH/LINK/LIST when CONTROL OP B4700 is *not* specified.

2620 Variable length records can only be specified if the hardware type is TAPE.

2621 EDIT mask and receiving field must be UA.

2622 Illegal controllers on BCT address.

2623 Illegal combination of move and/or arithmetic overrides.

2624 Overrides on a FILL statement must be DGT, CHR, or WDS.

2625 Either the address or the length is incompatible with the override specified.

2626 An attempt has been made to equate a data name to a label.

2627 A one address "snapshot" DUMP may not use indirect addressing.

2628 An attempt has been made to "ADDRESS =" to a label.

2629 COMMON relative data allowed in ICM only.

2630 An audited file must be HPT with work area access.

2631 Variable clause in FILE statement may only contain optional numeric literal.

2632 Mix function statements specifying an MCP name (example: MIX RUNNING) may not be used in the same program with mixed function statements not specifying an MCP name (example: MIX).

27 Limit Exceeded

2701 EDIT mask is greater than 100 characters long.

2702 Limit exceeded in dynamic array in COMPILER for segment header. Recompile requesting more MEMORY.

2703 Limit exceeded on 40 sort keys to SORT intrinsic.

2704 Data in program over 100,000 digits when CONTROL EXTENDED not specified.

2705 Coding in program exceeds 300,000 digits when CONTROL EXTENDED not specified.

2706 A UA controller has forced the stated increment to exceed 999999 digits.

2707 Increment in controller added to segment relative address causes address to exceed 999999.

2708 SEARCH table too large to handle.

2709 Subscript literal variable added to ARRAY address exceeds 999999.

2710 SEARCH construct key exceeds 100 digits or characters in length.

2711 SEARCH increment exceeds 100 digits or characters.

2712 The VALUE parameters length exceeds 99 digits.

2713 The length exceeds 100 on 3 way move links, i.e., [XCH].

2714 Limit exceeded in dynamic table for attributes. Please recompile with more memory.

2715 Resultant MOD adjustment for segment starting exceeds 9999.

2716 The size of REMAINDER will exceed 100 due to the combined lengths of the divisor and the quotient.

2717 The size of the buffer area in the FILL statement exceeds 9999 units (digits, characters or words).

2718 An integer field of 6 digits long is being preset to an address being equal to or greater than 100,000, i.e., no room for hardware controller.

2719 Too many DEFINEs or DEFINE/identifier duplications. Recompile giving more memory. Note - it is possible that a "define loop" has been generated. (A = B = A, etc.) or (A = B#, B = A#, etc.)

2720 A declaration greater than 7 digits long is being preset to an address that is equal to or greater than 100,000 but the segment has not gone EXTENDED.

2722 Over 99 segment strings to a parametric DEFINE.

2723 The length of the field for translation exceeds 10000 digits or bytes.

2724 The REMAINDER is greater than 99 digits (including sign) for the given operands.

2725 The length of the field to be SCANed exceeds 100 digits or characters.

2726 More occurrences of "ADDRESS;" than "ADDRESS =".

2727 An overlay level greater than 99 has been requested.

2728 The area specified for "snapshot" DUMP is too large or the ending address is greater than 999999.

2729 Only one segment is allowed in an ICM.

2730 More than one procedure with no global code may not be an ICM.

2731 DISPLAY LINES may not specify more than 999 lines.

2732 The size of the receiving field for SPOMESSAGE may not exceed 9998 digits.

2733 Data name size greater than 99 digits will not create multiple moves.

2734 EDITOR record greater than 72 characters.

2735 Nesting level too deep to build dictionary. Compile with additional MEMORY.

2736 Only 100 subroutines allowed.

2737 Limit exceeded in dynamic table for wait options. Please recompile with more memory.

2738 Length of record size identifier exceeds six digits.

28 Missing Key Words

2800 Missing key word or data name in data communications construct.

2801 Missing key word "KEY" with RANDOM file in FILE declaration.

2802 Missing key word or data name in SCAN statement.

2803 Missing key word or data name in SEARCH LINK/DELINK statement.

2804 Missing key word or data name in SEARCH statement.

29 Compiler Errors

NOTE

2900 Group errors may appear at times. These are internal compiler errors. They may be caused by certain combinations of syntax errors causing recovery problems. Upon encountering a 29nn error, follow the following steps.

1. Fix any obvious syntax errors and recompile.
2. If still present, retain all documentation and report error to your Unisys Technical Representative.

30 Illegal Special Characters

3000 Illegal right bracket.

31 Illegal Pictures

3100 Invalid PICTURE characters or missing ")".

3101 Repeat part of PICTURE in error.

3102 Float characters not valid.

3103 Invalid sequence of PICTURE characters.

3104 "P" PICTURE character(s) in error.

3105 Sign error.

3106 PICTURE requires mask that exceeds hardware limit 100.

3107 Size specification error.

32 Expression Errors

3201	Stack operand overflow, too many operands. Break expression into smaller sub-expressions.
3202	Stack operator overflow, too many operators. Break expression into smaller sub-expressions.
3203	Operand stack underflow. Too many operands pulled from stack.
3204	Operator stack underflow. Too many operators pulled from stack.
3205	Unexpected EOF.
3214	Operand cannot be used in generating a bit test (BOT or BZT).
3215	Too many left parentheses.
3216	Relational used in an assignment statement, i.e., A := (B=C).
3217	Attempt to used result of operation as receiving field, i.e., (A+B) :=C.
3218	Illegal operator within expression.
3219	Illegal operand in arithmetic operation.
3223	Too many right parentheses.
3224	Attempt to use result of conditional expressions as a value, i.e., IF (A=B) = C.
3225	Cannot use literal with "OR" in conditional mode.
3226	Illegal operator.
3227	Illegal operand in COMPARE statement.
3229	Buffer name of buffer construct is greater than 28 characters.
3230	No literal within parentheses for buffer number construct.
3231	Non-numeric literal used in buffer number construct.
3232	Value of literal for buffer number construct is greater than 9.
3233	Missing right parenthesis on buffer number construct.
3234	Buffer number does not exist.
3240	Illegal INFL on operand when temporary field is used as receiving field.
3241	[FPT] or [REM] too late in expression. Move to beginning of expression.

APPENDIX D
INDEPENDENTLY COMPILED MODULES (ICM)

The BPL compiler can generate Independently Compiled Modules (ICMs) which can be bound together with other modules to create a single program.

An ICM compilation creates a pseudo code file, rather than executable code. The binding process is required to create the executable program code.

Three types of ICMs can be created:

- Type I ICMs are for use in FORTRAN (ANSI 66) programs as functions or sub-routines. They must be bound into a FORTRAN program by the FORTIV or XFORTN compilers.
- Type II ICMs can be bound to other BPL Type II ICMs to create a single BPL program. They must be bound with the BPLBND binder program.
- Type III ICMs can be bound to any other Type III ICMs using the BINDER program. Type III ICMs can be produced by the FORT77, COBOL74, PASCAL and BPL compilers. For more information about the BINDER program, see the V Series BINDER Programming Reference Manual.

NOTE
BPLBND is a class C product.

TYPE I ICMs

Type I Independently Compiled Modules (hereafter referred to as ICM1s) can be created by the BPL compiler for inclusion in a subsequent FORTRAN compilation as as subroutine or a function.

An ICM1 compilcation causes a pseudo code file of the ICM1 portion to be built by the compiler. No code file is generated for that compile, and in fact, the only portion of the program compiled is any bounded by @ICM. See the @ICM construct in Compiler Directing Statements, Section 6, for further information.

An ICM1 may be either a block or a procedure, therefore, it may not contain more than one segment. Only one ICM1 may be created from a program segment, but several ICMs may be created during the compilation of a program. Only one ICM can be compiled ay any one time.

ICM1s may be called from within other ICM1s. See SUBROUTINE declaration and SUBROUTINE call statement.

There are three methods of communication between a calling program and an ICM1. These are as follows:

- Parameters
- COMMON blocks
- LINKAGE Construct

These are described on the following pages.

Parameters

These must be name parameters and care must be taken to ensure the agreement of the actual and formal parameters at the time of the CALL. FORTRAN does not check.

The following FORTRAN program calls a subroutine to manipulate the variables I, J, and A. I contains integer data, J is an integer variable containing character data, and A is a character variable.

```
IDENT     ICMTST
          CHARACTER*3 A /"ABC"/
          I = 1
          J = "DEF"
          CALL ICMEX (I, A, J)
          WRITE (6, 90) I, A, J
      90  FORMAT (1X, I7, 2X, A3, 2X, A3)
          STOP
          END
```

FORTIV and XFORTN have different data lengths and addressing considerations (see following), so the BPL declarations vary:

FORTIV Subroutine (Extended addressing and accumulator functions)

```
ICMEX:  BEGIN
CONTROL OP B4700, EXTENDED;  & EXTENDED is required
@ICM "ICMEX"
PROCEDURE  ADDONE (INT, CHAR1, CHAR2);
  FIXED INTEGER INT;  & Use FIXED for all numeric parameters
  ALPHA CHAR1  (3);   & CHARACTER*3 is 3 bytes
  ALPHA CHAR2  (3);   & Only want 3 bytes
  BEGIN
    INT    := INT + 1;  & Add 1 to INT
    CHAR1  := "UVW";    & Change contents of CHAR1
     & FORTIV passes numeric name params with UN
     & controllers.
     & Since CHAR2 has alpha data, change controller to UA
     & (2) CHAR2 is 3rd parameter, so its 8-digit address
     & is offset 32 digits into the stack:
    BASE.IX3.+32.1.UN := 2;  & Put UA controller on address
     & Now CHAR2 will be treated as alpha data:
    CHAR2 := "XYZ";   & Change contents of CHAR2
    EXIT;             & Return to FORTRAN
  END;
  @ICM
END;
```

Because FORTRAN permits alphabetic data to be stored in numeric fields, FORTIV replaces SN controller digits with UN controllers for the following data types. INTEGER, REAL, COMPLEX. This requires special handling in a BPL subroutine:

Signed Numeric Items

- BPL must declare these FIXED (in which case the signed controller is preserved and no adjustment is needed); or
- BPL must adjust the controller digit in the stack; or
- BPL must receive that type of item into an signed INTEGER field which is large enough to contain the sign digit. That field can then be moved to a second unsigned field which is redefined as SIGNED and is the field actually manipulated. The preceding example would then be in part:

```
PROCEDURE  ADDONE (INT);
INTEGER INT (8);
BEGIN
    INTEGER Y1 (8);
    ADDRESS = Y1;
      SIGNED INTEGER Y2 (7);
    ADDRESS;
    Y1  := INT;
    Y2  := Y2 + 1;  & Add 1 to INT
    INT := Y1;
    EXIT;
```

Alphanumeric Data in Numeric-Type Fields

BPL must adjust the controller in the stack (as with CHAR2 in the preceding example); or

BPL must receive the item into an unsigned INTEGER field which is redefined as ALPHA, and manipulate the redefined field (as in case (3) under Signed Numeric Fields).

FORTIV Subroutine Parameter Controller Digit Rules Summarized

Signed controller digits are preserved for DOUBLE data types and for data types declared FIXED.

Alphanumeric controller digits are preserved for the FORTRAN data types CHARACTER and LOGICAL.

Other types of fields have unsigned numeric (UN) controller digits.

FORTIV Stack Parameters and Extended Addresses

FORTIV always places 8-digit extended addresses into the stack. Thus, the BPL routine must be compiled with CONTROL EXTENDED, and the BPL programmer must remember that parameter addresses in the stack are offset from each other by eight digits.

XFORTN Subroutine

```
ICMEX: BEGIN
@ICM "ICMEX"
PROCEDURE  ADDONE (INT, CHAR1, CHAR2);
  SIGNED INTEGER INT(5);            & XFORTN integer is 5 bytes +
                                    sign
  ALPHA CHAR1 (3);                  & CHARACTER*3 is 3 bytes
  ALPHA CHAR2 (3);                  & Integer length is 3 bytes
  BEGIN
    INT    := INT + 1;              & Add 1 to INT
    CHAR1 := "UVW";                 & Change contents of CHAR1
     & XFORTN passes numeric items with SN controllers.
     & Since CHAR2 has alpha data, change controller to UA
     & CHAR2 is 3rd parameter, so its 6-digit address is
     &   offset 28 digits into the stack:
    BASE.IX3.+28.1.UN := 2;         & Put UA controller on address
     & Now CHAR2 will be treated as alpha data
    CHAR2 := "XYZ";                 & Change contents of CHAR2
    EXIT;                           & Return
  END;
  @ICM
END;
```

FORTRAN permits alphanumeric data to be stored in numeric fields, but XFORTN places a signed numeric (SN) controller digit on all numeric fields. When alphanumeric data is passed as a parametr, the BPL programmer must follow the directions for Alphanumeric Data in Numeric-Type Fields, as described with the preceding FORTIV subroutine.

XFORTN Subroutine Parameter Controller Digit Rules Summarized

Signed (SN) controller digits are preserved for all numeric data types.

Alphanumeric (UA) controller digits are preserved for the FORTRAN data types CHARACTER and LOGICAL.

XFORTN Stack Parameters

XFORTN uses only 6-digit (non-extended) addresses. Thus, the BPL subroutine must NOT declare CONTROL EXTENDED. Stack parameter addresses are offset from each other by six digits.

COMMON Blocks

When ICM1s are loaded by the FORTRAN loader, it matches up the names of COMMONs and maps the variables onto the COMMON area (FORTRAN) in the specified sequence. To declare these variables in a BPL generated ICM1, a COMMON block is coded. There may be a maximum of 100 COMMON blocks, each with a unique COMMON name. The format is:

COMMON common-name BEGIN common-declarations END ;

The COMMON declaration types between compilers match as follows:

FORTIV with Storage Unit Mapping*	BPL
INTEGER	SIGNED INTEGER (7) MOD 4, ALPHA (2)* FIXED INTEGER, ALPHA (2)*
REAL	REAL (8) MOD 4 FIXED REAL
DOUBLE	REAL (16) MOD 4, ALPHA (2)* FIXED DOUBLE, ALPHA (2)*
LOGICAL	INTEGER (2) MOD 4, ALPHA (5)*
CHARACTER*nn	ALPHA (nn) MOD 4

NOTE

*Under Storage Unit mapping, a numeric item is mapped into the minimum number of Numeric Storage Units required to contain the item. A numeric Storage Unit is 6 bytes (12 digits) of storage, word aligned. When smaller items (INTEGER or LOGICAL) occupy a Numeric Storage Unit, the Storage Unit is padded with trailing blanks, which the BPL program must take into account. CHARACTER*nn data is stored in nn Character Storage Units (CSU). A CSU is one byte and word aligned. The character string is padded with a trailing blank to a word boundary, if necessary.

FORTIV without Storage Unit Mapping* ($NOSU)	**BPL**
INTEGER	SIGNED INTEGER (7) FIXED INTEGER
REAL	REAL (8) FIXED REAL
DOUBLE	REAL (16) FIXED DOUBLE
LOGICAL	INTEGER (2)
CHARACTER*NN	ALPHA (NN)

XFORTN	**BPL**
INTEGER	SIGNED INTEGER (integer size)
REAL	REAL (real size) SIGNED INTEGER (real size +3)
DOUBLE	REAL (double size) SIGNED INTEGER (double size +3)
LOGICAL	INTEGER (2)
CHARACTER*NN	ALPHA (NN)

FORTRAN manuals contain further information about COMMON and its uses.

LINKAGE Construct

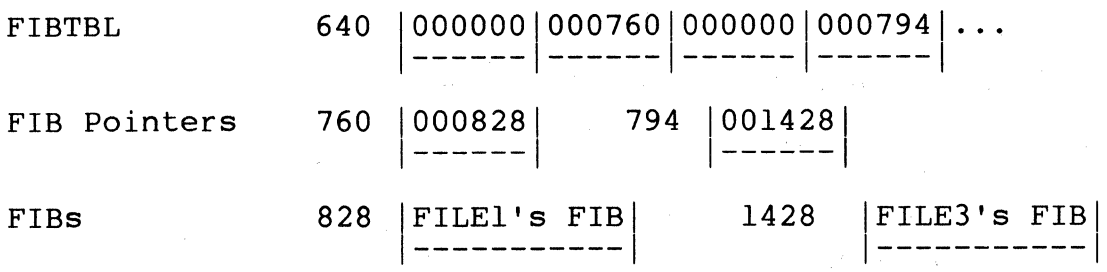
The LINKAGE construct is useful only when writing your own FORTRAN intrinsics or subroutines which must access files, or XFORTN external functions. LINKAGE provides an interface much like COMMON, with a fixed mapping. Figure D-1 describes the format of the LINKAGE declaration.

```
LINKAGE identifier-1 [ , . . . , identifier-9]  ;
```

Figure D-1. Format of the LINKAGE Declaration

The identifier contain the addresses of the following fields, in this order:

1. FIBTBL — Address of the program's FIB table (constructed by the FORTRAN compiler). Table contains 20 entries, numbered 0 to 19. Each entry contains the address of a pointer to the FIB for the correspondingly-numbered file. If an entry is zero, no file with that number was declared. The example below shows the relationship between FIBTBL-to-FIB. The program in this example has declared FILE1 and FILE3; addresses shown are examples only. The number preceding the boxed data is an address in the program's memory. The boxed data represents the contents of memory at that address.



   The address of a file's FIB can thus be obtained by using the file number as an index into the FIBTBL.

2. FRESULT — Function result field (XFORTN only).

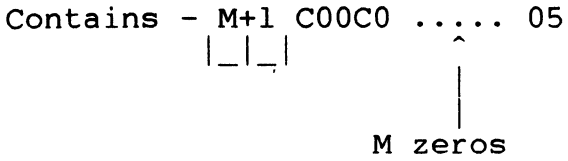
   Size - max (2x(M+4), 2L, N=1)

3. INTRNTMP — Intrinsic temporary.

   Size - 4X(M+4)

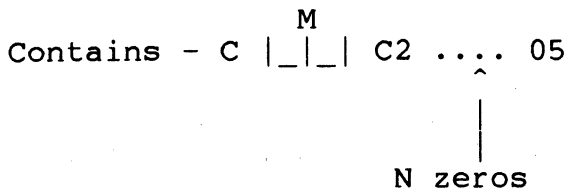
4. HALFADJ — Half adjust constant.

   Size - M + 7



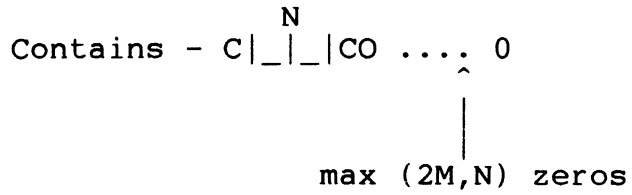
5. TWOCNST — A constant of two used for Integer to Real conversion when Integer size is greater than Real size.

   Size - N + 5



6. NFLOATFLD — Used for Real to Integer conversion.

   Size - max ((2xM)+4, N+4)



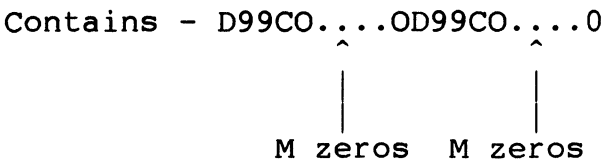
7. ACCUMFLD — Temporary storage

   Size = (2xM) +4

   Contains - zeros

8. CMPLXZERO — Complex zero

   Size - 2 X (M+4)



9. Not used.

NOTE

M = Real size; N = Integer size; L = Alpha size

Example 1 - File handling

The following routine could be called from a FORTIV program to close a tape file with no rewind and then reopen the file with no rewind. The file number is passed in the CALL: CALL TAPMK (file-number).

```
BEGIN:
CONTROL OP B4700, EXTENDED;
@ICM "TAPMK"
PROCEDURE TAPMK (TAPENO);
FIXED INTEGER TAPENO;
BEGIN
    LINKAGE FIBTBL;
    OWN FIXED INTEGER X;
    OWN INTEGER       FIBADDR (6);
    INTEGER           TAPEUNIT (6);
     & Store file number in TAPEUNIT for use as index
    X         := TAPENO;
    TAPEUNIT := X.+2.6.UN;          & Bypass Extended Index/Controller
                                    & Get address of FIB
    IX2 := TAPEUNIT * 6;            & 6 digits per FIBTBL entry
    IX2 := FIBTBL.IX2.6.UN;         & IX2 now has address of FIB pointer
    FIBADDR := BASE.IX2.6.UN;       & Put FIB address into FIBADDR
     & At this point the FIB's address is in FIBADDR. The file
     &  is accessed by using FIBADDR as an indirect address.
     &  This sample routine will merely close and open the file.
    CLOSE FIBADDR.IA NO REWIND;
    OPEN FIBADDR.IA NO REWIND;
    EXIT;
END;
@ICM
END:
```

Example 2 - BPL External Function for FORTRAN

The following program invokes a function that increments the argument of the function by one:

```
IDENT   FNCALL
        I = 0
        I = INC (I)
        WRITE (6, 90) I
   90   FORMAT (1X, I6)
        STOP
        END
```

FORTIX and XFORTN have different methods of returning function results. When using a BPL ICM for XFORTN, the result must be left in the second field of the LINKAGE area (the Function Result Field). For FORTIV, the result must be left in the accumulator.

XFORTN Function

```
FUNCTN:  BEGIN
@ICM "INC"
PROCEDURE  INC(X)
  SIGNED INTEGER X(5)
  BEGIN
      LINKAGE FIELD1, FRESULT;
      FRESULT.5.SN := X + 1;  & Increment and store result
      EXIT;
  END;
@ICM
END;
```

Length and type overrides must be specified for FRESULT.

FORTIV Function

```
FUNCTN:  BEGIN
CONTROL OP B4700, EXTENDED;
@ICM "INC"
PROCEDURE  INC(X);
   FIXED INTEGER X;
   BEGIN
      IACCUM := X + 1;          & Increment and store result in accum
      EXIT;
   END;
@ICM
END;
```

FORTRAN ICM Considerations

You must observe the following rules when you develop BPL ICMs for FORTIV/XFORTN programs:

- You must enter multi-procedure ICMs at the last procedure.
- Procedures must not contain other procedures.
- Files must be declared in the first procedure.

TYPE II AND TYPE III ICMs

The BPL compiler can create Independently Compiled Modules of the Type II or Type III form. Type III ICMs can only be created by a BPL compiler in the 2.*n* series of software releases. BPL compilers in the 7.*n* series of releases cannot create Type III ICMs.

One or more Type II ICMs (ICM2s) can be bound (using the free-standing binder program BPLBND) to create a single executable program. One or more Type III ICMs (ICM3s) can be bound into an executable program by using the BINDER program.

A Type II ICM is created when the dollar option $ICM2 is set in the leading dollar record in a compile. This option must be on a leading dollar record and cannot be reset. (The @ICM directive used to create Type I ICMs is not used for Type II ICMs.)

A Type III ICM is created when the dollar option $ICM3 is set in the leading dollar record in a compile. This option must be on a leading dollar record and cannot be reset. The @ICM directive is not used.

No code files are created for any ICM2s or ICM3s being compiled. A code file is only created when one or more ICM2s are bound together or when one or more ICM3s are bound together.

ICM2s or ICM3s can contain a single procedure or multiple procedures.

BPL Language Constructs for Type II and Type III ICMs

Type II and Type III ICMs are not compatible and cannot be bound together. The BPL language constructs used to create Type II and Type III ICMs, however, are the same for either type of ICM. The following constructs are implemented in the BPL compiler for use with Type II and Type III ICMs (ICM2/3):

a. Module Name Declaration:

   Names the code module and the ICM2/3 file.

b. Program Entry Point Declaration:

   Specifies that the module in which it is declared contains the procedure which is the program entry point, and names that procedure.

c. Entry Declaration:

   Specified a procedure in this ICM2/3 which can be called from a procedure in another ICM2/3 and specified the parameters involved, if any.

d. External Declaration:

   Specifies a procedure in another ICM2/3 which is being called from within this ICM2/3, gives the local and external names of the procedure, and describes the parameters involved, if any.

These constructs are explained further in the following pages.

NOTE
Generation of Type III ICMs is only available under BPL 2.0 or greater.

MODULE NAME DECLARATION

The first source record of an ICM2/3 must appear as it is shown in Figure D-2.

```
module-name: BEGIN
```

Figure D-2. First Source Record of an ICM2/3

Example

```
ICMBL1: BEGIN      & Begins an ICM named ICMBL1
```

The module-name identifies the code contained within the ICM2/3 file.

In addition, the first six characters of the module name become the default library name of the compiled ICM2/3 file, unless overidden by a BPL compiler FILE equate statement when the ICM2/3 is compiled:

Type II ICM:
?CMP<prog name>BPL;FILE ICM2FL=<ICM2 file name>
Type III ICM:
?CMP<prog name>BPL;FILE ICM3FL=<ICM3 file name>

If required, PROG_ENTRY, ENTRY, and EXTERNAL declarations must appear immediately following this source record.

PROGRAM ENTRY POINT DECLARATION

The function of a PROG_ENTRY declaration is to specify that this ICM2/3 contains the procedure which is the program entry point and to indicate that procedure's name.

Figure D-3 shows the format of the PROG-ENTRY declaration.

```
PROG ENTRY = procedure-name ;
```

Figure D-3. Format of the PROG-ENTRY Declaration

One PROG_ENTRY declaration must be specified for each bound program.

Procedure-name is required and may be up to 30 characters in length. This is the name of the procedure which begins the bound program.

The entry point of a bound program is the first instruction in the procedure indicated in the PROG_ENTRY statement. This differs from a conventional, unbound BPL program; its entry point is the first executable instruction following the procedure decarations. This follows from the the language restriction that all executable code in the outer block must be within procedures.

ENTRY DECLARATION

The function of an ENTRY declaration is to specify that this ICM2/3 contains a procedure that is called by another ICM2/3. It also describes the structure of the parameters to be passed, if any. Figure D-4 describes the format of ENTRY.

```
ENTRY = procedure-name
        [ (formal-parameter-list) ;
        L VALUE value-parameter-list ; ]
        parameter-specifications] ;
```

Figure D-4. Format of ENTRY

Example:

```
ENTRY = PROG3(A,B);          & PROG3 is called from another
  ALPHA A(10);               &   ICM2/3. It has two ALPHA
        B(10);               &   parameters, A and B
```

A separate ENTRY declaration is required for each procedure that is called by another ICM2/3.

The syntax for this statement following the equal sign is identical to a FORWARD PROCEDURE declaration.

Procedure-name is required and can contain up to 30 characters. This is the name of the procedure which is called by another ICM2/3.

The formal-parameter-list is optional. If specified, it names the formal parameters of the procedure. Up to 10 parameters may be specified, must be separated by commas, and must be enclosed by parentheses.

The VALUE clause is optional and can only be used if parameters are specified. The word VALUE must be followed by one or more of the named formal parameters in the formal-parameter-list that are VALUE parameters. The parameters must be separated by commas.

The parameter-specifications are required if parameters are specifed. There must be one specification for each formal-parameter-list. VALUE parameters are so declared in a separate VALUE declaration list prior to any formal parameter declarations. The specification identifier is to be the same as the one used in the formal-parameter-list. The specification identifier is to be the same as the one used in the formal-parameter-list.

EXTERNAL DECLARATION

The function of an EXTERNAL declaration is to indicate that a procedure in another ICM2/3 is to be called from this ICM2/3. The EXTERNAL statement also equates the local name used for the external pocedure in the call to the procedure-name given in the ICM2/3 in which that procedure is declared; this permits a procedure to be called by a name other than the name by which it was declared. If any parameters are involved, they are specified.

Figure D-5 describes the format of the EXTERNAL declaration.

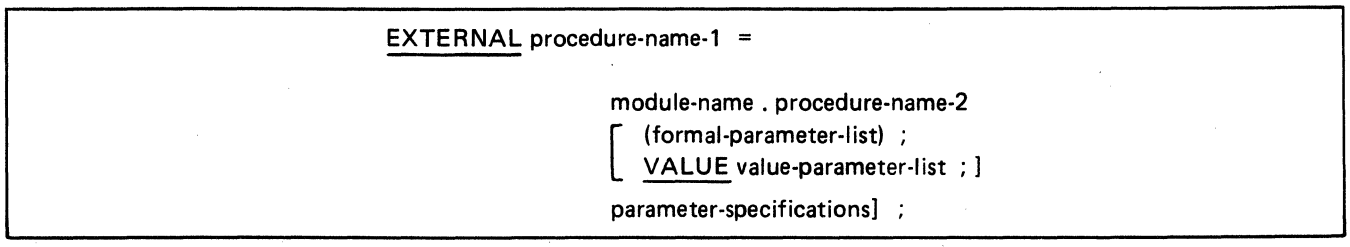


Figure D-5. Format of the EXTERNAL Declaration

Example:

```
EXTERNAL PROG2 =          & A procedure in this ICM2/3
    ICMBL2.PROG3(A,B);    &   will call
                              PROG2, which
  ALPHA A(10),            &   is declared in the ICM2/3
        B(10);            &   "ICMBL2" as procedure
                          &   "PROG3", with ALPHA
                          &   parameters A and B.
```

A separate EXTERNAL declaration is required for each procedure in another ICM2/3 which is called by a procedure in this ICM2/3.

This statement replaces any SUBROUTINE declarations.

Procedure-name-1 is required and can contain up to 30 characters. This is the local name of a procedure in another ICM2/3 which is being called from this ICM2/3. Procedure-name-1 must be the procedure name used in the procedure call in this ICM2/3, excluding parameters.

The module-name is required. If the module-name specified is longer than six charactes, only the first six will be used. This name is used to identify the appropriate ICM2/3 which contains the procedure being called.

Procedure-name-2 is required and can contain up to 30 characters. This is the name of the procedure being called as it appears in its ICM2/3; it must be written exactly as it appears in its ICM2/3, including parameters.

Module-name and procedure-name-2 must be separated by a period. No embedded blanks are allowed.

The formal-parameter-list is optional. If specified, it names the formal parameters of the procedure. Up to 10 parameters may be specified, must be separated by commas, and must be enclosed by parentheses.

The parameter-specifications are required if parameters are specified. There must be one specification for each formal parameter in the formal-parameter-list. VALUE parameters are so declared in a separate VALUE declration list prior to any formal parameter declarations. The specification identifier is to be the same as the one used in the parameter list.

An EXTERNAL procedure cannot be a user label routine (see compiler restriction b, Programming Considerations, in this manual).

Programming Considerations for Type II and Type III ICM

A Type II ICM is created when the dollar option $ICM2 is present in a compile. A Type III ICM is created with the dollar option $ICM3. Either option must be on a leading dollar record and neither option can be reset. (The @ICM directive used to create Type I ICMs is not used for Type II or Type III ICMs.)

No code file is created for any Type II or Type III ICM (ICM2/3) being compiled. A code file is only created when one or more ICM2/3s are bound together.

An ICM2/3 can contain a single procedure or multiple procedures.

All segmentation directives such as SEGMENTED and UNSEGMENTED are treated as noise words; segmentation is determined at bind time.

Following are language restrictions that apply when you create ICM2/3s:

- All Global data will be allocated into a special named COMMON block called <module-name>.GLOBAL.

  NOTE
  This allocation is done by the compiler.

- Any executable code within the outer must be within procedures.

  This means that the program entry point in a bound program will be at a procedure. This differs from a conventional, unbound BPL program, which is entered at the first instruction following procedure declarations.

- All procedures within procedures will be considered as part of their outer procedure and cannot be called by another ICM2/3 or at bind time.

- To use dynamic storage, a COMMON block named "page_space_info" (the name must be in lower-case letters) must be declared. The declaration must consist of three INTEGER fields, each six digits in length.

  When the program is bound, the first field will be initialized to contain the address of the beginning of the dynamic area (the memory space beyond that used for instructions and data, and preceding the base of the program stack).

  An execution time, code supplied by BPLBND or BINDER (depending on the type of ICM; Type II or Type III) will store the program's total memory size in a field called "memory_bct_response". This field is allocated by BPLBND or BINDER (depending on the type of ICM; Type II or Type III); the programmer need not declare the field. The supplied code will then initialize the remaining fields in "page_space_info" to the addresses of the base of the stack and limit of the program. The dynamic storage area will be a fixed size.

- Global labels are not allowed.

- Bits cannot be passed as parameters to procedures.

The following compiler restrictions apply to ICM2/3s.

- When producing Type II or Type III ICMs, do not use the dollar-card option CODE. Program code, summary, and segmentation information can be obtained from BPLBND (Type II) or the BINDER (Type III).
- In a FILE declaration, if the ROUTINE option is used, the user label routine cannot be declared EXTERNAL.
- In a data declaration, the data name cannot be preinitialized to the address of a procedure name (for example, INTEGER (6) A := [PROC1]), if the procedure name is declared EXTERNAL.

The following additional compiler restrictions apply when creating Type II or Type III ICMs: attempting to code any of them will result in a compiler error message number 2912:

- The "ADDRESS OF" a procedure name in a different segment cannot be assigned; that is:

  A := [procedure-name]

  will work only if A and procedure-name are in the same segment.
- In a data declaration, if the ROUTINE option is used, the user label routine cannot be declared EXTERNAL.

There is no compatibility between Type I, Type II ICMs or Type III ICMs, nor is any intended. Type I ICMs can only be bound by FORTIV and XFORTN; Type II ICMs can only be bound by BPLBND. Type III ICMs can only be bound by the BINDER. The different ICM types cannot be mixed in any bind.

Example

In this example three Type II ICMs are created and bound into a single program. For an example of binding Type III ICMs, see the V Series BINDER Programming Reference Manual.

The program shown in the following figures is intended to illustrate the use of language constructs for Type 2 ICMs, and not to constitute a practical application. As such, its functions are limited.

The first module contains a single procedure, which is the main procedure for the program and is designated as the program's entry point. This procedure calls procedures located in the two other modules to manipulate variables and to print information.

Two COMMON blocks are in the first module. One holds two variables and will be accessed by one of the EXTERNAL procedures. The other illustrates the use of a "page_space_info" declaration to obtain information about the dynamic storage area.

The second module contains a single procedure and one of the COMMON blocks declared in the first module. It calls a procedure from the third module to list the variables in the COMMON block, and then manipulates the variables.

The third module contains three procedures and a FILE declaration. Two of the procedures can be called to open and to close the file. The third procedure writes an output record constructed from a series of ALPHA parameters. Refer to Figures D-1, D-2, and D-3.

The ICM2s can be bound together as program-id TESTER by BPLBND specifications such as the following:

```
?COMPILE TESTER WITH BPLBND LIB
?DATA INPUT
REQUIRED MODULES:
    ICMBL1 FROM ICMBL1,
    ICMBL2 FROM ICMBL2,
    ICMBL4 FROM ICMBL4;
STACKSIZE = 300;
END;
?END
```

See the BPLBND section for further information on program binding, and for an example of another way to bind this program.

```
ICMBL1_ICM: BEGIN
    PROG_ENTRY = PROG1;                    & Declare entry point
    EXTERNAL PROG_2 =                      & Declare
      ICMBL2_ICM.PROG3;                    &   all
    EXTERNAL OPENIT =                      &   referenced
      ICMBL4_ICM.OPENIT;                   &   procedures
  EXTERNAL WRITIT =                        &   which
      ICMBL4_ICM.WRITIT-prog(A,B,C):       &   are
      ALPHA A(12),                         &   external
            B(12),                         &   to
            C(12);                         &   this
  EXTERNAL CLOSIT =                        &   module
      ICMBL4_ICM.CLOSIT;                   &
  CONTROL OP B4700, EXTENDED;
  ALPHA PRIX (12);                         & work
  ADDRESS = PRTX;                          &   area
    NUMERIC PRT_NR (c);                    &   for
    ALPHA   PRT_FILL (6) :=" ";            &   passing
  ADDRESS;                                 &   numbers
PROCEDURE PROG1:
  BEGIN
    COMMON page_space_info                 & "page-space-info" fields
      BEGIN                                &   are pre-initialized:
        INTEGER DYNAM_BASE (6),            &   --by BPLBND
                STACK_BASE (6),            &   --by BPLBND-supplied
                PRGM_LIMIT (6);            &   --  code
      END;
  &
    COMMON COM1                                & Used
      BEGIN                                    &   by
        ALPHA X(12),                           &   PROG1
              Y(12);                           &   and PROG_2
      END;
  &
    OPENIT;                                    & Open printer file
    X := "AAAAAAAAAA";                         & Initialize fields
    Y :+ "BBBBBBBBBB";                         &   in block COM1
  &
    WRITIT ("PROG1","  X = ", X);              & List original X
    PROG_2;                                    & Interchange X & Y
    WRITIT ("PROG1", "NEW X = ", X);           & List replaced value
```

Figure D-6. Type II ICM Example, First Module

```
  &
    PRT_NR := DYNAM_BASE;                        & List
    WRITIT ("PROG1", "DYNAM_BASE =",PRTX)        &  values
    PRT_NR := STACK_BASE;                        &  contained in
    WRITIT ("PROG1", "STACK_BASE =",PRTX);       &  the COMMON
    PRT_NR := PRGM_LIMIT;                        &  block
    WRITIT ("PROG1", "PRGM_LIMIT",PRTX);         &  "page_space_info"

  &
    CLOSIT;                                     & Close printer file
  END;
END;


COMPILE DATE 18:05  05/15/79  USING 010/79 BPL.  PROGRAM ID IS ICMBL1.

ELASPED TIME      7 SECONDS.  RELEASE NUMBER:  ASR 6.2

ELASPED TIME IS TOTAL CLOCK TIME, NOT TIME CHARGEABLE TO COMPILATION.

    52 SYMBOLIC RECORDS COMPILED AT  445 RECORDS PER MINUTE.

TOTAL NUMBER OF ERROR MESSAGES IS    0.

TOTAL NUMBER OF WARNING MESSAGES IS  0 INCLUDING NO  SEQUENCE ERRORS.
```

Figure D-6. Type II ICM Example, First Module (Continued)

```
ICMBL2_ICM: BEGIN
  ENTRY = PROG3;                          & Declare entry point
  EXTERNAL WRIT_IT =                      & Declare
      ICMBL4_ICM.WRITIT_proc(A,B,C);      &   referenced
      ALPHA A(12),                        &   procedure
            B(12),                        &   external to
            C(12);                        &   this module
  CONTROL OP B4700, EXTENDED;
PROCEDURE PROG3;
  BEGIN
  &
      COMMON COM1                         & Used
        BEGIN                             &   by this
          ALPHA X(12),                    &   procedure
                Y(12);                    &   and PROG1
        END;
  &
      WRIT_IT ("PROG_2","  X = ", X)     & List variables
      WRIT_IT ("PROG_2","  Y = ", Y); &   at entry
  &
      X := Y [XCH];                       & Interchange variables
  END;
END;
```

```
COMPILE DATE 18:05  05/15/79  USING 010/79 BPL.  PROGRAM ID IS ICMBL2.

ELASPED TIME   6 SECONDS.  RELEASE NUMBER:  ASR 6.2

ELASPED TIME IS TOTAL CLOCK TIME, NOT TIME CHARGEABLE TO COMPILATION.

    23 SYMBOLIC RECORDS COMPILED AT   230 RECORDS PER MINUTE.

TOTAL NUMBER OF ERROR MESSAGES IS   0.

TOTAL NUMBER OF WARNING MESSAGES IS  0 INCLUDING NO  SEQUENCE ERRORS.
```

Figure D-7. Type II ICM Example, Second Module

```
ICMBL4_ICM:  BEGIN
  ENTRY = OPENIT;                          & Declare
  ENTRY = WRITIT_proc (A,B,C);             &  entry
      ALPHA A(12),                         &  points
            B(12),                         &  into
            C(12);                         &  this
  ENTRY = CLOSIT;                          &  module
  CONTROL OP B4700, EXTENDED;
  ALPHA PRNREC (132);                      & Output
  ADDRESS = PRNREC,                        &  record
      ALPHA PRT1 (12),                     &  for
            PRT2 (12),                     &  printer
            PRT3 (12);                     &  file
  ADDRESS;                                 &
  FILE ICMPRT, PRINTER, RECORD PRNREC 132:
       &
PROCEDURE OPENIT;                          & Open the
  BEGIN                                    &  printer
      OPEN OUT ICMPRT;                     &  file and
      PRNREC := [ALL] " ";                 &  clear
  END;                                     &  record
       &
PROCEDURE WRITIT_proc (A,B,C);             & Assemble
      ALPHA A (12),                        &  and
            B (12),                        &  write
            C (12);                        &  an output
      BEGIN                                &  record
          PRT1 := A;                       &  with contents
          PRT2 := B;                       &  passed from
          PRT3 := C;                       &  another
          WRITE ICMPRT PRNREC;             &  procedure;
          PRNREC := [ALL] " ",             &  clear record
      END;                                 &
        &
PROCEDURE CLOSIT;                          & Close
      BEGIN                                &  the
          CLOSE ICMPRT RELEASE;            &  printer
      END;                                 &  file
END;
```

Figure D-8. Type II ICM Example, Third Module

THE BPLBND PROGRAM BINDER

BPLBND is a stand-alone program which creates a single executable code file from one or more BPL Type II ICMs. BPLBND is not valid for Type III ICMs, which are bound together with the BINDER program.

The programmer or program which calls BPLBND indicates which Type II ICMs contribute to the code file.

The programmer or program which calls BPLBND indicates the segmentation layout of the bound program.

Code files written by BPLBND may be loaded and executed by the MCP.

Functional Description

BPLBND accepts control statements and Type II ICMs as input. As output, an executable code file (executable program) is produced along with a diagnostic listing of the bind. Optionally a code listing is provided at the diagnostic listing.

A complete binder execution deck follows the form:

```
                                                    [ { LIB     } ]
                                                    [ { LIBRARY } ]
                                                    [   SAVE    ]
                                                    [   SYNTAX  ]
 ?{CMP    }
  {COMPILE}<code-file-name> [WITH] BPLBND

 ?DATA INPUT.
 Input statements:
   Selection
   Option
   Segmentation
   END Statement
 ?END
```

The <code-file-name> field on the COMPILE statement indicate the external or directory name of the bound code file; that is, the name of the executable program being procduced.

BPLBND Input Statements

The BPLBND input statements indicate which Type II ICMs to include in the bind, and describe the overlay structure.

Input specifications are free-format and are separated from one another by semicolons. Multiple statements may appear on one line, and a single statement may extend onto several lines.

A "line" is character positions 1 through 72 of an 80-character record. Positions 73 through 80 are reserved for sequence numbers.

Comments may be entered following a "%". BPLBND ignores all characters between a "%" and column 72, inclusive.

The elements ina list are separated from one another by commas. No word or number may straddle two source images or contain blanks.

Binder keywords must be upper-case. Names may be lower-case, upper-case, or both. Since there is no folding of lower-case to upper-case, case conventions established in the BPL modules for files names, module names, and block names must be followed.

BPLBND accepts four classes of input statements.

1. Selection Statements:

   Selection statements specify the required modules and optional modules for the bind. The first required modules named is the host module.

   - REQUIRED statement

     The REQUIRED statement names those modules which must be included in the bound program. The first required module named is the host module.

   - OPTIONAL statement

     This statement lists optional modules. An optional module is included in the bound program only if there is a reference to a procedure in the optional module from a module which has been included.

2. Option Statements

   Option statements control error classifications, code file address extension, printer listings, and code file memory and stack size.

   The option control statements are:

   - PRINTANALYSIS, PRINTCODE, PRINTREFERENCES, PRINTSEGANALYSIS, and PRINTALL statement

     These statements determine what information is printed on the bind listing.

   - PROGRAMLIMIT, PROGRAMSIZE, and STACKSIZE

     These statements regulate program and stack size for the bound code file.

   - NOEXTEND statement

     Specifing NOEXTEND prevents the use of extended addresses in the bound code file.

   - FATAL statement

     The level at which errors prevent formation of a code file can be reset with the FATAL statement.

3. Segmentation Statements

   Segmentation statements describe the overlay structure of the bound program.

   - SEGMENT statement

     This statement names the blocks that will be bound into segments.

   - OVERLAY statement

     This statement specifies the overlay structure of the bound program.

4. Terminator Statement

   The END statement indicates the end of the input specifications.

A module must be named in a selection statement before it can be named in an option or segmentation statement.

A description of each of these input statements follows, in the order given previously.

BPLBND INPUT SELECTION STATEMENTS:

REQUIRED Statement

The REQUIRED Statement names those modules (and type II ICMs) that must be included in the bind. Figure D-9 describes the format of this statement.

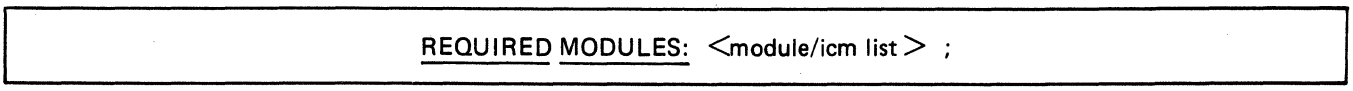


Figure D-9. Format of the REQUIRED Statement

An item in a <module/icm list> is of the form:

<module name> FROM <icm name>

Items in this list are separated by commas.

The REQUIRED statement must immediately precede the OPTIONAL statement, if used.

A module is required when it contains at least one procedure or COMMON block which must be included in the bound program.

The first module named is the host module.

In the event that no explicit program entry point has been specified in any of the contributing Type II ICMs, the first entry point in the host becomes the default program entry point.

Example:

```
REQUIRED MODULES:
MODULA     FROM     ICM001,
MODB       FROM     MODB,
MODC       FROM     MODC;
```

This statement specifies that MODULA, MODB, and MODC are required components of the program being bound, and that they are found in the ICM2 files named ICM001, MODB, and MODC, respectively.

(It is only necessary to name the module containing the program entry, the host module, in a REQUIRED statement. All other modules may be declared OPTIONAL if the programmer wishes, since they will be included if called).

OPTIONAL Statement

The OPTIONAL statement names those modules (and Type II ICMs) that will be used to satisfy unresolved external references from the REQUIRED modules. OPTIONAL modules are included in the bound program only if they are called from other modules that have been included in the bound program. Figure D-10 shows the format of the OPTIONAL statement.

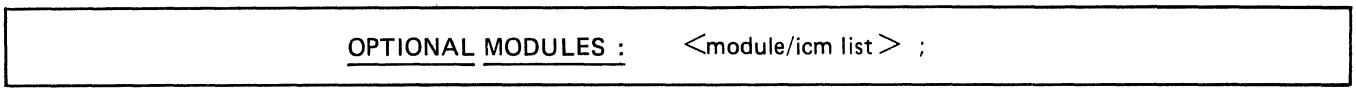


Figure D-10. Format of the OPTIONAL Statement

An item in a <module/icm list> is in the following form:

<module name> FROM <icm name>

Items in this list are separated by commas.

The OPTIONAL statement may be used to name modules which can be called by alternate versions of a conditionally compiled program. This permits the use of a single set of BPLBND control statements which can be used for all versions of a conditionally compiled program. Only the modules actually called by the version compiled will be included in the bound program.

Example:

```
OPTIONAL MODULES:
MODX    FROM    ICMOOX,
MODY    FROM    MODY,
MODZ    FROM    MODZ;
```

This statement specifies that MODX is to be bound into the program if there is a reference to it from another bound module. Likewise, either or both of MODY and MODZ will be bound modules. The three modules are to be found in the ICM2 files named ICMOOX, MODY, and MODZ, respectively.

BPLBND OPTION STATEMENTS

FATAL Statement

The FATAL statement resets the level at which errors become fatal (that is, prevent the formation of a code file). Figure D-11 shows the format of the FATAL statement.

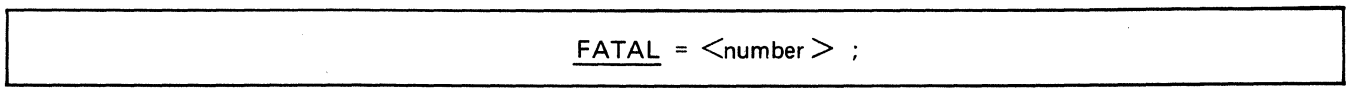


Figure D-11. Format of the FATAL Statement

<Number> is a number between 1 and 10.

Errors are ranked by severity level; the default FATAL level is 6. Level 10 errors are always fatal.

A FATAL statement takes effect as soon as it is recognized. All BPLBND input statements are parsed before any other processing is done. When used, a FATAL statement would normally be the first BPLBND input statement.

Normally, the FATAL level would not be reset. A programmer may use this statement to force a code file to be generated despite the detection of certain errors by BPLBND, or to cause low-level errors to prevent formation of a code file.

NOEXTEND Statement

The NOEXTEND statement prevents creation of a code file which uses extended addressing. Figure D-12 describes the format of this statement.

```
NOEXTEND ;
```

Figure D-12. Format of the NOEXTEND Statement

BPLBND PRINT STATEMENT

The PRINT statements specify optional information to be printed on the BPLBND output listing, following the diagnostic output and in addition to the minimum information provided.

The various PRINT options are described on the followng pages. They are all of the general form shown in Figure D-13

<Print Statement> [<print list>] ;

Figure D-13. General Format of PRINT Options

The <Print Statements>s are: PRINTALL, PRINTANALYSIS, PRINTCODE, PRINTREFERENCE, AND PRINTSEGANALYSIS.

On this and the following pages, a "code block" is a BPL procedure, and a "data block" is a BPL COMMON block.

When no PRINT statement is specified, the minimum printed output from BPLBND is a diagnostic listing of the input statements, a list of the ICM2s included in the bind and their compile dates, a map of the program's overall memory layout, base addresses and sizes of the code and data blocks present, the address of the host module's entry point, and a summary giving the number of fatal and non-fatal errors.

The <print list> is defined as follows:

```
[ { <module name>                                                    } ]
[ { <module name>   . <procedure>                                    } ] [,.....]
[ { <module name>   { <procedure-1>,...,<procedure-n> }              } ]
```

If every code block is required, omit the print <print list>.

Any explicitly referenced modules must previously have been named in a selection statement.

PRINTALL Statement

The PRINTALL statement names those (code) blocks for which both a code listing and an analysis must be printed. Figure D-14 describes the format of the PRINTALL statement.

```
PRINTALL  [<print list>] ;
```

Figure D-14. Format of PRINTALL

The <print list> is described under PRINT Statements.

PRINTALL combines the effects of the PRINTANALYSIS and PRINTCODE statements.

<Print list>s from PRINTALL statements add to the cumulative <print list>s from both PRINTANALYSIS and PRINTCODE statements.

For example, the following statement causes code and analysis listings to be printed for all of MODULE, for PROC1 in MODB, and for P1, P2, and P3 in MODC:

```
PRINTALL MODULE, MODB.PROC1, MODC(P1,P2,P3);
```

PRINTANALYSIS Statement

The PRINTANALYSIS statement names those (code) blocks for which analyses must be printed. Figure D-15 describes the format of this statement.

```
PRINTANALYSIS  [ <print list> ]  ;
```

Figure D-15. Format of PRINTANALYSIS

The analysis provides a list (an inverse reference graph) indicating the code blocks which access the blocks specified in the <print list>. That is, for each specified block, a list is provided of all code blocks which access that block.

<Print list>s from separate PRINTANALYSIS statements are cumulative.

The <print list> is described under PRINT Statements.

If every block is required, omit the <print list>. A reference graph of the entire program is then provided.

Any explicitly referenced modules must have been named previously in a selection statement.

For example, the following statement causes analysis listings to be printed for all of MODULE, for PROC1 in MODB, and for P1, P2, and P3 in MODC:

```
PRINTANALYSIS MODULE, MODB.PROC1, MODC(P1,P2,P3) ;
```

Output from PRINTANALYSIS appears on the BPLBND listing under the heading "REFERENCED BLOCKS". The blocks to be analyzed, and the code blocks referring to each, are listed. Blocks which are not explicitly accessed by user program statements in any procedure are marked "NOT REFERENCED."

Output lines take the following form:

```
{ CODE }
{ DATA }   BLOCK <name>  SEG #n  { NOT REFERENCED                          }
                                 { REFERENCED BY                           }
                                 {   CODE BLOCK <name> SEG #n [ . . . ]    }
```

PRINTCODE Statement

The PRINTCODE statement names those (code) blocks for which a code listing must be printed. Figure D-16 describes the format of this statement.

PRINTCODE [<print list>] ;

Figure D-16. Format of PRINTCODE

<Print list>s from separate PRINTCODE statements are cumulative.

The <print list> is described under PRINT Statements.

If every code block is required, omit the <print list>.

Any explicitly referenced modules must have been named previously in a selection statement.

For example, the following statement causes code listings to be printed for all of MODULE, for PROC1 in MODB, and for P1, P2, and P3 in MODC.

```
PRINTCODE MODULA, MODB.PROC1, MODC(P1,P2,P3);
```

PRINTSEGANALYSIS Statement

The PRINTSEGANALYSIS statement indicates that intersegment references must be printed. Figure D-17 describes the format of this statement.

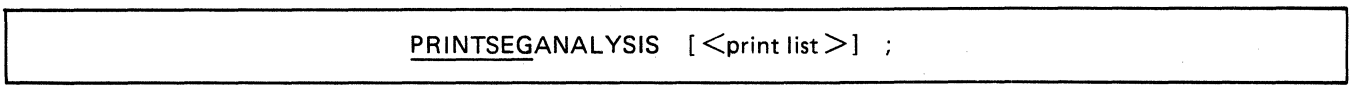


Figure D-17. Format of PRINTSEGANALYSIS

The <print list> is described under PRINT Statements.

If every code block is required, omit the <print list>.

Any explicitly referenced modules must have been named previously in a selection statement.

For example, the following statement causes intersegment references to be listed for all of MODULA, for PROC1 in MODB, and for P1, P2, and P3, in MODC:

```
PRINTSEG MODULA, MODB.PROC1, MODC(P1,P2,P3);
```

Output from the PRINTSEGANALYSIS statement appears on the BPLBND listing under the following heading:

```
REFERENCES MADE BY PROCS OF SEGMENT #n: <segment-name>
```

Procedures in segment #n and the blocks accessed are listed in the same format as the output from PRINTREFERENCES. Only inter-segment references are listed; references between blocks in the same segment will not appear. If both PRINTREFERENCES and PRINTSEGANALYSIS are specified in the BPLBND control statements, only the output from PRINTSEGANALYSIS will appear on the listing.

PROGRAMLIMIT Statement

This statement specifies the maximum desired size of the program. Its format is shown in Figure D-18.

```
PROGRAMLIMIT = <max 6 digit number> ;
```

Figure D-18. Format of PROGRAMLIMIT

If the size specified is exceeded, a warning message is provided.

PROGRAMSIZE Statement

This statement specifies the minimum size (in digits) of the bound program. Its format is shown in Figure D-19

```
PROGRAMSIZE = <max 6 digit integer> ;
```

Figure D-19. Format of PROGRAMSIZE

The program will be expanded to the specified size if it would otherwise be smaller. If the program is larger then the specified size, a warning message is provided.

STACKSIZE Statement

This statement permits the user to specify the size of the stack. Figure D-20 shows the format of this statement.

```
STACKSIZE = <max 6 digit number> ;
```

Figure D-20. Format of STACKSIZE

If this statement is not present, the size assigned is the largest of the stack values provided by the ICM2s.

BPLBND SEGMENTATION STATEMENTS

SEGMENT Statement

A SEGMENT statement names the blocks within the input modules which are to be bound together into a (possibly overlayable) segment. The "data blocks" shown in Figure D-21 are BPL COMMON blocks; "CODE blocks" are procedure blocks.

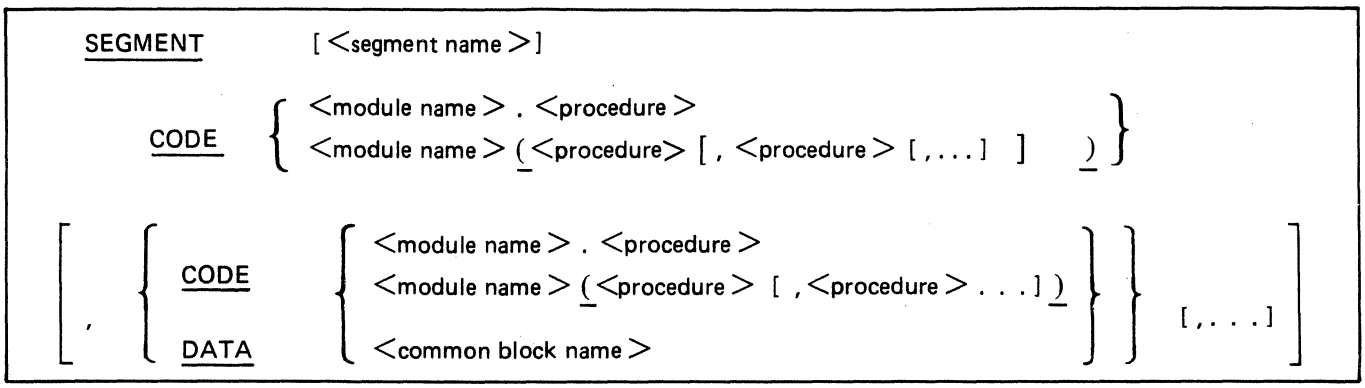

SEGMENT [<segment name>]
CODE { <module name> . <procedure>
<module name> (<procedure> [, <procedure> [, . . .]]) }
[, { CODE { <module name> . <procedure>
<module name> (<procedure> [, <procedure> . . .]) }
DATA { <common block name> } } [, . . .]]

Figure D-21. SEGMENT with Data Blocks

The <segment name> names the composite segment created from the code and the data items listed after it.

Procedure names are preceded by the word CODE. COMMON block names are preceded by the word DATA.

A segment consists only of the code and data blocks explicitly assigned to it in a SEGMENT statement, and any associated constant pools (see following). A DATA block local to a CODE block will be placed in the same segment as the code only if both are named in the same SEGMENT statement.

Constant and address constant pools for a procedure will be placed in that procedure's segment. These areas will be identified in BPLBND output listings as the following:

<module name>.<procedure name>.CONST_POOL
<module name>.<procedure name>.ACON_POOL

If a <segment name> is duplicated, the <segment group> for the first occurrence is the one recognized.

A <segment name> can appear only once in the OVERLAY statement of a bind.

Example:

```
SEGMENT   X
  CODE    PROGRM1 . PROCEDURE1 ;
```

This creates a segment called X. The segment will contain the procedure called procedure_1, from the module called PROGRM1.

```
SEGMENT    J
  CODE   PROGRM1 (E, F, G) ;
```

This statement will cause procedures E, F and G from the module PROGRM1 to be contained in a segment called J.

```
SEGMENT    Y
  CODE   (...),
  DATA   PROGRM1.COM1 ;
```

This statement creates segment Y, containing the COMMON block COM1 from the module PROGRM1, as well as the CODE block(s) specified. (An overlayable segment must contain at least one code block or it will never be brought into memory).

OVERLAY Statement

The OVERLAY statement determines which code and data blocks are overlayable, which segments share the same addresses and, hence, overlay one another, and where segments begin in memory relative to one another.

Define the OVERLAY statement as shown in Figure D-22.

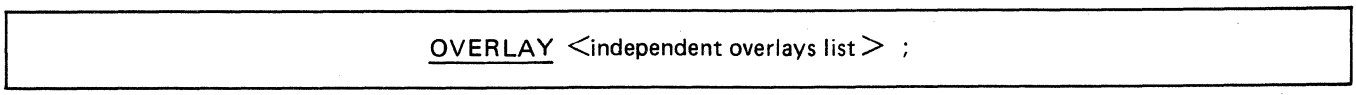


Figure D-22. Definition of the OVERLAY Statement

The <independent overlays list> is a list of independent overlays (defined below) separated by commas. Each <independent overlay> is specified as a list of one or more concurrent groups (defined below) separated by the symbol "<."

A single <independent overlay> is specified as follows:

```
{<segment name>                          }   <   {<segment name>                          }
{(<independent overlays list>)           }       {(<independent overlays list>)           }
[...<{<segment name-n>                   }  ...]
     {(<independent overlays list -n>)   }
```

For example, a valid OVERLAY statement would be:

```
OVERLAY A < (B,C), D, J < (H,I)
          (1)       (2)     (3)
```

In this example, the groups labelled (1), (2), and (3) are independent overlays (defined below). Within group (1), A and (B,C) are concurrent groups (defined below); withih group (3), J and (H,I) are concurrent. In turn, B and C within (1) are independent; H and I within (3) are independent.

Program memory is divided into 2 areas: one for overlayable code data segments and one for non-overlayable (resident) segments. The OVERLAY statement describes the segmentation structure for the overlayable code data area. Any segment not indicated in an OVERLAY statement is non-overlayable by default.

The layout of items in the program's overlayed region is hierarchically described by subclusters of items which are either "independent" or "concurrent".

Independent groups are never simultaneously active in the program, and hence can occupy the same storage addresses at different points in time. Such groups share the same base address. The longest group determines the end address of the region.

Concurrent groups are those which need to be present in memory simultaneously for program efficiency and hence, must occupy disjoint ranges of addresses. Such a group is a "working set". The end address of one group becomes the base address of the next group.

Any block units are formed by the concatenation of code and data blocks. These overlays can be defined and named in SEGMENT statements.

Any block included in a bind but not listed in an OVERLAY statement is placed at an arbitrary position in resident memory.

It is possible to include a code block in several overlays; a separate copy will bw created for each instance. There can be only one copy of a data block in a program.

A <segment name> can appear only once in the OVERLAY statement.

If there is no OVERLAY statement, the entire program will be located in resident memory.

In the previous example,

```
OVERLAY A < (B,C), D, J < (H,I)
```

where J consists of the procedures E and F. Figure D-23 shows how G (as specified in a SEGMENT statement) is mapped.

Base of Overlayed Region	End of Overlayed Region	Possible Working Sets
A	B	AB
	C	AC
D		D
E F G	H	EFGH
	I	EFGI

Figure D-23. Mapping of Overlayed Regions

BPLBND TERMINATOR STATEMENT

END STATEMENT

The END statement, which is optional, marks the end of the binder input statements. Figure D-24 describes the format of this statement.

```
END ;
```

Figure D-24. Format of the END Statement

If the END statement is omitted, a warning is provided.

Input-Output Facilities of BPLBND

BPLBND reads two kinds of input files: a card image file containing BPLBND input statements (these input statements must name one or more Type II ICM files), and the ICM2 files that will be bound.

BPLBND writes two kinds of output files. It always generates a printer backup file which echoes BPLBND input statements, lists all bind-time errors, gives the overall memory map of the program and addresses of code and data blocks, and optionally prints code and analysis listings for selected code blocks. If there are no bind-time errors above the FATAL level, BPLBND create an executable code file.

Table D-1 lists BPLBND input and files.

Table D-1. BPLBND Input and Files

File Description	Internal	External
Input statements	INPUT	INPUT
Diagnostic/code listing	PRINT	PRINT

Many BPLBND output message are printed in lower-case letters. Users whose printeres lack lower-case characters will need to use the TRN option of the MCP's PRINT (or PB) command when printing PBLBND output listings, to fold such messages into upper-case.

Debugging and Diagnostic Facilities of BPLBND

BPLBND provides extensive error reporting and control, bind-time parameter checking, and extended addressing veto power.

CODE AND DATA INFORMATION, ADDRESSES AND REFERENCES

Compile dates of the bound ICM2s, an overall program memory map, base addresses and sizes of code and data blocks, and the host module's entry address are always provided by BPLBND. See the PRINT statements for additional information whch can be obtained.

PARAMETER CHECKING

Each inter-module entry point definition or reference has an ICM2 data structure which describes the formal or actual parameters. Calling and receiving sequences are compared to this description and must have the same number of parameter digits and use the same kind of addresses for the call by reference parametes.

ERROR HANDLING

BPLBND recognizes two classes of error: fatal and non-fatal. It reports each instance of each error found in the lasting for the bind.

No code file is produced if a fatal error is detected. Whenever possible, BPLBND completes the current processing phase after it finds a fatal error so that other errors may be reported. It does not proceed to its next processing phase.

An executable code file is produced if no fatal errors are encountered. Errors are ranked on a scale from 1 to 10. The higher the number, the more serious the error. Ten is always fatal. A FATAL statement makes BPLBND treat all errors of the severity level given and greater as fatal errors. The defalult FATAL level is 6.

It is the programmer's responsibility to determine the impact a BPLBND detected error will have on the generated code to decide whether to permit that error to remain.

All error messages are identified by a number. The following list describes the various possible errors by error number.

Error	Level	Error/Disposition
101	1	Number expected. Value for fatal error level is unchanged.
102	1	Unrecognized statement. All characters up to the next semicolon are skipped.
103	1	Keyword expected: xxxx BPLBND keeps trying.
104	1	Improper punctuation: xxxx name expected. All characters up to the next semicolon are skipped.
105	1	Segment xxxx undefined. This segment is ignored.
107	1	FATAL = xxxx: value exceeds 10. Value for fatal error level is unchanged.
109	1	Punctuation expected: xxxx. BPLBND keeps trying.
110	5	Segment xxxx duplicated in OVERLAY statement.
111	1	END statement expected. End of input encountered; statement fragment is ignored.
112	1	Identifier too long: remaining characters truncated. The left-most characters are processed and the remaining characters are skipped.
113	1	Duplicate OVERLAY statement. The previous OVERLAY statement is ignored.
114	5	Datablock xxxx.xxxx appears in more than one segment.
115	10	No HOST indicated. Fatal. The bind cannot proceed beyond syntax checking without a host module.
116	10	xxxx out of space. Fatal. BPLBND has overflowed the named in-memory table.
118	5	Module xxxx is not in module list.

Error	Level	Error/Disposition
119	1	Identifier expected. Value for fatal error level is unchanged.
120	1	Number too long. The left-most digits are processed and the rest ignored.
121	1	Module names xxxx and xxxx from ICM xxxx do not match. Value for fatal error level is unchanged.
122	1	Illegal character. Value for fatal error level is unchanged.
123	1	CODE or DATA expected. All characters up to the next semicolon are skipped.
124	1	Segment xxxx is not in OVERLAY statement.
190	5	xxxx appears in more than one Type II ICM. The first Type II ICM name is used. Fatal. A target name for a module cannot be determined.
203	5	Unresolved reference from command deck to xxxx. The block was named in the command deck but did not appear in a Type II ICM.
204	5	No code block in bind is named xxxx. The external reference is unresolved. It is linked to a binder-supplied routine that will issue a run-time message and terminate program execution at that point.
205	10	File not found. Fatal. A file named in a binder input statement was unavailable when BPLBND tried to open it.
206	10	Interface error.

Error	Level	Error/Disposition
207	10	Total calling sequence length different in definition and reference. Fatal. Code is not safe if number of digits placed on stack differs from number of digits read from stack in a procedure call.
208	10	Incompatible return seg_on_stackfield. Fatal. Called PROC does not allow room for "return segment number" on stack frame.
209	10	Incompatible address extension. Fatal. Modules use different representations of call-by-reference parameters.
213	6	Reference parameter list is shorter/longer than definition parameter list. The definition is linked with the refernce.
214	3	Lengths are incompatible for parameter xxxx. The definition is linked with the reference.
215	3	Types are incompatible for parameter xxxx. The definition is linked with the reference.
216	3	Classes are incompatible for parameter xxxx. The definition is linked with the reference.
218	5	Program entry points are not included in the bind. Fatal.
221	10	Module xxxx is of incorrect ICM format for this BPLBND release. Fatal.
222	5	Cannot find module xxxx. Fatal.
223	1	To xxxx.xxxx from xxxx. This message qualifies other messages.
303	2	This block was named in segment xxxx.

Error	Level	Error/Disposition
304	5	Segment xxxx cannot be reused. The second and subsequent occurrences of segment xxxx in an overlay list are skipped rather than expanded.
305	2	Data block xxxx is read_write and cannot be overlayed.
402	7	Incompatible descriptions for data block xxxx-description from first module chosen. BPLBND uses the first description encountered as a data block.
407	10	Incorrect index to code block in ICM xxxx. The block is ignored.
408	5	ICM error - illegal data block in module xxxx. The block is ignored.
409	10	Reference out of bound in module xxxx. Fatal.
501	10	ICM error - entry point index out of range.
502	10	ICM error - interface index out of range.
503	10	ICM error - no interface table.
600	10	Block xxxx mapped above 100KD limit.
601	10	Block xxxx mapped over 300KD limit. Fatal.
602	6	Program above 100KD but extensions vetoed.
603	5	xxxx cannot be duplicated.
604	5	xxxx must be resident.
605	10	xxxx has BCT over 300KD. Fatal.
606	1	Program larger than specified size.
701	10	Invalid controller on a never extended branch instruction. Fatal.

Error	Level	Error/Disposition
702	10	Absolute address of never extended address form at exceeds 300000. Fatal.
703	10	Absolute address of never extended address form at exceeds 100000. Fatal.
704	10	Invalid code token type. Fatal.
705	10	Invalid read on ICM.
706	10	Trouble on LABEL TABLE file. Fatal.
707	10	Trouble on CODE file. Fatal.
709	10	Displacement offset exceeds the limit of the block. Fatal.
710	10	Absolute address is less than zero. Fatal.
711	10	Invalid dynamic destination. Fatal.
712	10	Error in file. Fatal.
713	4	ICM already opened in error.
801	1	Field previously initialized xxxx.
802	10	Host ICM is not available. Fatal.
803	4	Host ICM was already opened.
804	5	Invalid OP code in print table.
805	10	Invalid segment number index.
806	10	An address is > 6 digits. Fatal.

Error	Level	Error/Disposition
940	10	Invalid write while copying LABEL TABLE from xxxx. Fatal.
976		Binder error in summary.
999		Not implemented.

Operational Considerations for BPLBND

There are several size restrictions in BPLBND:

- There can be at most 99 Type II ICMs.
- There can be at most 99 segments.
- There can be at most 20 files.
- In the entire program there can be at most 9998 (code and data) blocks, 9998 references to code or non-local data, 998 entry points, 9998 external references, and 9998 parameters.
- In a single Type II ICM there can be at most 200 blocks and 200 external references.
- The memory requirement of the bound program (after segmentation) must not exceed 300 KD.

In binding a large program, performance of BPLBND can be enchanced by adding additional memory (?MEM + 20 following the compile command).

BPLBND Examples

Example 1:

This example illustrates a complete set of BPLBND control statements. A hypothetical program is created from procedures located in several ICM2s.

```
?COMPILE TST2BK WITH BPLBND LIB
?FILE PRINT = TST2BK PBK FORM
DATA INPUT

FATAL = 9;
STACKSIZE = 2000;
REQUIRED MODULES:
  prt_typ2_icm    FROM II22bk,
  ranrd           FROM IB22bk,
  stop            FROM X5ESTO,
  err             FROM X5ESER,
  print           FROM X5EPRN,
  readcd          FROM X5EREA;

OPTIONAL MODULES:
  debug           FROM X5EBDB,
  trace           FROM X5ETRC,
  dump            FROM X5EDMP,
  param           FROM X5EPAR,
  arm             FROM X5EARM,
  trap            FROM X5ETRP,
  put             FROM X5EPUT;

PRINTSEG;

SEGMENT driver
  CODE prt_typ2_icm.main;

SEGMENT parser
  CODE prt_typ2_icm(mark_error, echo,
  next_char,  advance_to,
  get_ident,  get_number,
  next_token);

SEGMENT header
  CODE prt_typ2_icm(print_header, print_day_and_time);
```

```
SEGMENT blocks
  CODE prt_type2_icm(print_all_block_des,
  print_block;
  prt_block_descriptor);

SEGMENT eoj
  CODE err.error,
  CODE stop.program;

OVERLAY driver, parser, header, blocks, eoj;
END;
?END
```

The BPLBND printer output file ID will be TST2BK.

The FATAL statement has been used to increase the level number at which errors become fatal from 6 to 9.

The size of the bound program's stack will be 2000 digits.

The modules "prt_typ2_icm," "ranrd," "stop," "err," "print," and "readcd" are required for this program. That is, each module contains at least one procedure or COMMON block which must be bound into the program.

The modules "debug," "trace," "dump," "param," "arm," "trap," and "put" contain procedures or COMMON blocks which can or cannot be called by any of the other modules bound into the program.

Intersegment references will be printed on the diagnostic listing.

Table D-2 shows how the five segments are declared.

Table D-2. Declaring the Five Segments

Segment Name	Explanation
driver	Contains the procedure "main" from the module "prt_typ2_icm."
parser	Contains the procedure "mark_error," "get_number," and "next_token" from the module "prt_typ2_icm."
header	Contains the procedures "print_header" and "print_day_and_time" from "prt_typ2_icm."
blocks	Contains the procedures "print_all_block_desc," "print_block," and "prt_block_descriptor" from the module "prt_typ2_icm."
eoj	Contains the procedure "error" from the module "err," and the procedure "program" from the module "stop."

The five segments are declared to be independent overlays. That is, all begin at the same address when in memory; no two will be in memory simultaneously.

Finally, the END statement terminates the input.

Example 2:

The illustrations that follow show BPLBND control statements used to bind the three ICM2s given in the example at the end of the section on BPL Type II ICMs. The printed output produced by BPLBND is also shown.

All modules are named in the REQUIRED list. Only ICMBL1_ICM, which contains the program entry point, must be named there; the others can be named as OPTIONAL MODULES, and BPLBND determines that they must present.

Code, analysis, and reference listings are requested. A segment analysis listing is not requested, because that would suppress the PRINTREFERENCES list.

Three segments are specified. The first, main_seg, contains the main procedure and the COMMON data block COM1. The procedures called to open and to close the output files are grouped into a second segment, and the procedures called to write output messages and to manipulate the variables in COM1 are in the third segment.

Since the second and third segments are never in use at the same time, they are declared to be independent and able to overlay each other.

The program stack size is specified to be 300 digits.

The remaining figures show the information provided by BPLBND. Bound modules and their compile dates are always listed. Warnings follow when BPLBND has not found references to certain blocks. These warnings can be ignored when they do not name blocks explicitly used in the modules.

A map of the program's overall memory layout is always produced. When segmentation is requested, the segments and their locations and sizes are listed, and a diagram of the spatial relationships between the segments is given.

Details of addresses and sizes of all code and data blocks within all segments are always given. This listing shows the locations in resident memory of all blocks not named in SEGMENT statements, as well as the final layout of all segments declared in SEGMENT statements.

A list of modules entry points is given.

The output from the PRINTREFERENCES option is next. A heading is printed for every segment. Each procedure within the segment is listed, and the code and data blocks to which instructions in the procedure refer are listed.

```
EXECUTION OF BINDER (Version 7/13/78 A)

STACKSIZE = 300;

REQUIRED MODULES:
  ICMBL1_ICM  FROM  ICMBL1,
  ICMBL2_ICM  FROM  ICMBL2,
  ICMBL4_ICM  FROM  ICMBL4;

PRINTREF;
PRINTCODE;
PRINTANALYSIS;

SEGMENT main_seg
  CODE ICMBL1_ICM.PROG1,
  DATA COM1;

SEGMENT open_close
  CODE ICMBL4_ICM.OPENIT,
  CODE ICMBL4_ICM.CLOSIT;

SEGMENT work_seg
  CODE ICMBL4_ICM.WRITIT_proc,
  CODE ICMBL2_ICM.PROG3;

OVERLAY  main_seg < (open_close, work_seg);

END;
```

Figure D-25. BPLBND Example, Control Statement Listing

```
   SEGMENT LAYOUT:

   ---SEGMENT NAME---  SEG    BASE    SIZE    SLACK    INTERCEPT CODE
   RESIDENT              1     260    1636        0           44
   main_seg              2    1896     536        0          140
   open_close            3    2432      80      160            0
   work_seg              4    2432     240        0            0


SEG  LO---------------SPATIAL RELATIONSHIPS-------------------------HI
  2  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  3                                        XXXXXXXXX----------------
  4                                        XXXXXXXXXXXXXXXXXXXXXXXXXX
```

Figure D-26. BPLBND Example, Program Information Listing (Sheet 1 of 8)

```
----MODULE NAME----  ICM NAME---------COMPILED BY----COMPILE DATE

     ICMBL1_ICM      ICMBL1     BPL                      5/15/79 18: 5
     ICMBL2_ICM      ICMBL2     BPL                      5/15/79 18: 5
     ICMBL4_ICM      ICMBL4     BPL                      5/15/79 18: 6
****203 UNRESOLVED REF FROM COMMAND DECK TO iheap (LEVEL 5: WARNING)


OVERALL MEMORY LAYOUT:

----REGION-----            BASE    SIZE
RESERVED MEMORY               0     100
SEG DICTIONARY              100     160
RESIDENT SEG                260    1636
OVERLAY AREA               1896     776
PAGE SPACE                 2672    1028
STACK                      3700     300
TOTAL PROGRAM                 0    4000
```

Figure D-26. BPLBND Example, Program Information Listing (Sheet 2 of 8)

```
DETAILS OF SEGMENT #1:  RESIDENT

      BASE     SIZE   EXT    ----------BLOCK NAME----------------
                 0             DATA BLOCK pparam
                 0             DATA BLOCK BRANCH_TARGET
       488      18             DATA BLOCK page_space_info
       508       8             DATA BLOCK memory_bct_response
       516      24             DATA BLOCK ICMBL1_ICM.GLOBAL
       540     280             DATA BLOCK ICMBL1_ICM.PROG1.PROG1
                 0             DATA BLOCK ICMBL2_ICM.GLOBAL
       820      96             DATA BLOCK ICMBL2_ICM.PROG3.PROG3
       916     264             DATA BLOCK ICMBL4_ICM.GLOBAL
      1180     672             DATA BLOCK ICMPRT


DETAILS OF SEGMENT #2:  main_seg

      BASE     SIZE   EXT    ----------BLOCK NAME----------------
      1896     348             CODE BLOCK ICMBL1_ICM.PROG1
                 0             DATA BLOCK ICMBL1_ICM.PROG1.CONST_POOL
                 0             DATA BLOCK ICMBL1_ICM.PROG1.ACON_POOL
      2244      48             DATA BLOCK COM1

DETAILS OF SEGMENT #3:  open_close

      BASE     SIZE   EXT    ----------BLOCK NAME----------------
      2432      48             CODE BLOCK ICMBL4_ICM.OPENIT
                 0             DATA BLOCK ICMBL4_ICM.OPENIT.CONST_POOL
                 0             DATA BLOCK ICMBL4_ICM.OPENIT.ACON_POOL
      2480      32             CODE BLOCK ICMBL4_ICM.CLOSIT
                 0             DATA BLOCK ICMBL4_ICM.CLOSIT.CONST_POOL
                 0             DATA BLOCK ICMBL4_ICM.CLOSIT.ACON_POOL


DETAILS OF SEGMENT #4:  work_seg

  BASE     SIZE     EXT     ----------BLOCK NAME----------------
  2432      128             CODE BLOCK ICMBL4_ICM.WRITIT_proc
              0             DATA BLOCK ICMBL4_ICM.WRITIT_proc.CONST_POOL
              0             DATA BLOCK ICMBL4_ICM.WRITIT_proc.ACON_POOL
  2560      112             CODE BLOCK ICMBL2_ICM.PROG3
              0             DATA BLOCK ICMBL2_ICM.PROG3.CONST_POOL
              0             DATA BLOCK ICMBL2_ICM.PROG3.ACON_POOL
```

Figure D-26. BPLBND Example, Program Information Listing (Sheet 3 of 8)

```
MODULE ENTRY POINTS:

------MODULE NAME------ ----PROCEDURE NAME----  SEG  ADDRESS
ICMBL1_ICM                 PROG1                  2     1896
ICMBL2_ICM                 PROG3                  4     2560
ICMBL4_ICM                 OPENIT                 3     2432
ICMBL4_ICM                 WRITIT_proc            4     2432


REFERENCES MADE BY PROCS OF SEGMENT #1:  RESIDENT


REFERENCES MADE BY PROCS OF SEGMENT #2:  main_seg

PROC ICMBL1_ICM.PROG1 SEG #2 REFERS TO
                             CODE BLOCK ICMBL4_ICM.OPENIT SEG #3
                             DATA BLOCK ICMBL1_ICM.PROG1.PROG1 SEG #1
                             DATA BLOCK COM1 SEG #2
                             CODE BLOCK ICMBL4_ICM.WRITIT_proc SEG #4
                             CODE BLOCK ICMBL2_ICM.PROG3 SEG #4
                             DATA BLOCK page_space_info SEG #1
                             DATA BLOCK ICMBL1_ICM.GLOBAL SEG #1
                             CODE BLOCK ICMBL4_ICM.CLOSIT SEG #3


REFERENCES MADE BY PROCS OF SEGMENT #3:  open_close

PROC ICMBL4_ICM.OPENIT SEG #3 REFERS TO
                             DATA BLOCK ICMBL4_ICM.GLOBAL SEG #1
PROC ICMBL4_ICM.CLOSIT SEG #3 REFERS TO NOTHING


REFERENCES MADE BY PROCS OF SEGMENT #4:  work_seg

PROC ICMBL4_ICM.WRITIT_proc SEG #4 REFERS TO
                             DATA BLOCK PROGRAM_RESERVED_MEMORY SEG #6
                             DATA BLOCK ICMBL4_ICM.GLOBAL SEG #1
                             DATA BLOCK ICMPRT SEG #1
PROC ICMBL2_ICM.PROG3 SEG #4 REFERS TO
                             CODE BLOCK ICMBL4_ICM.WRITIT_proc SEG #4
                             DATA BLOCK COM1 SEG #2
```

Figure D-26. BPLBND Example, Program Information Listing (Sheet 4 of 8)

```
   REFERENCED BLOCKS

DATA BLOCK pparam SEG #1 NOT REFERENCED
DATA BLOCK BRANCH_TARGET SEG #1 NOT REFERENCED
DATA BLOCK iheap SEG #1 NOT PRESENT NOT REFERENCED
DATA BLOCK page_space_info SEG #1 REFERENCED BY
                              CODE BLOCK ICMBL1_ICM.PROG1 SEG #2

DATA BLOCK memory_bct_response SEG #1 NOT REFERENCED
CODE BLOCK ICMBL1_ICM.PROG1 SEG #2 NOT REFERENCED
DATA BLOCK ICMBL1_ICM.PROG1.CONST_POOL SEG #2 NOT REFERENCED
DATA BLOCK ICMBL1_ICM.PROG1.ACON_POOL SEG #2 NOT REFERENCED
DATA BLOCK COM1 SEG #2 REFERENCED BY
                              CODE BLOCK ICMBL2_ICM.PROG3 SEG #4
                              CODE BLOCK ICMBL1_ICM.PROG1 SEG #2

CODE BLOCK ICMBL4_ICM.OPENIT SEG #3 REFERENCED BY
                              CODE BLOCK ICMBL1_ICM.PROG1 SEG #2

DATA BLOCK ICMBL4_ICM.OPENIT.CONST_POOL SEG #3 NOT REFERENCED
DATA BLOCK ICMBL4_ICM.OPENIT.ACON_POOL SEG #3 NOT REFERENCED
CODE BLOCK ICMBL4_ICM.CLOSIT SEG #3 REFERENCED BY
                              CODE BLOCK ICMBL1_ICM.PROG1 SEG #2

DATA BLOCK ICMBL4_ICM.CLOSIT.CONST_POOL SEG #3 NOT REFERENCED
DATA BLOCK ICMBL4_ICM.CLOSIT.ACON_POOL SEG #3 NOT REFERENCED
CODE BLOCK ICMBL4_ICM.WRITIT_proc SEG #4 REFERENCED BY
                              CODE BLOCK ICMBL2_ICM.PROG3 SEG #4
                              CODE BLOCK ICMBL1_ICM.PROG1 SEG #2

DATA BLOCK ICMBL4_ICM.WRITIT_proc.CONST_POOL SEG #4 NOT REFERENCED
DATA BLOCK ICMBL4_ICM.WRITIT_proc.ACON_POOL SEG #4 NOT REFERENCED
CODE BLOCK ICMBL2_ICM.PROG3 SEG #4 REFERENCED BY
                              CODE BLOCK ICMBL1_ICM.PROG1 SEG #2

DATA BLOCK ICMBL2_ICM.PROG3.CONST_POOL SEG #4 NOT REFERENCED
DATA BLOCK ICMBL2_ICM.PROG3.ACON_POOL SEG #4 NOT REFERENCED
DATA BLOCK ICMBL1_ICM.GLOBAL SEG #1 REFERENCED BY
                              CODE BLOCK ICMBL1_ICM.PROG1 SEG #2

DATA BLOCK ICMBL1_ICM.PROG1.PROG1 SEG #1 REFERENCED BY
                              CODE BLOCK ICMBL1_ICM.PROG1 SEG #2

DATA BLOCK ICMBL2_ICM.GLOBAL SEG #1 NOT REFERENCED
DATA BLOCK ICMBL2_ICM.PROG3.PROG3 SEG #1 NOT REFERENCED
DATA BLOCK ICMBL4_ICM.GLOBAL SEG #1 REFERENCED BY
                              CODE BLOCK ICMBL4_ICM.WRITIT_proc SEG #4
                              CODE BLOCK ICMBL4_ICM.OPENIT SEG #3

DATA BLOCK ICMPRT SEG #1 REFERENCED BY
                              CODE BLOCK ICMBL4_ICM.WRITIT_proc SEG #4
```

Figure D-26. BPLBND Example, Program Information Listing (Sheet 5 of 8)

```
CODE LISTING FOR CODE BLOCK ICMBL1_ICM.PROG1, SEG #2

02  001896   NTR    310002      002292
02  001908   CNST   0
02  001909   CNST   002
02  001912   MVA    101012      200540    202244
02  001930   MVA    101012      200560    202268
02  001948   NTR    310014      002336
02  001960   CNST   0
02  001961   CNST   002
02  001964   ACON   20000580
02  001972   ACON   20000604
02  001980   ACON   20002244
02  001988   NTR    310002      002380
02  002000   CNST   0
02  002001   CNST   002
02  002004   NTR    310014      002336
02  002016   CNST   0
02  002017   CNST   002
02  002020   ACON   20000628
02  002028   ACON   20000652
02  002036   ACON   20002244
02  002044   MVA    100608      000488    200516
02  002062   NTR    310014      002336
02  002074   CNST   0
02  002075   CNST   002
02  002078   ACON   20000676
02  002086   ACON   20000700
02  002094   ACON   20000516
02  002102   MVA    100606      000494    200516
02  002120   NTR    310014      002336
02  002132   CNST   0
02  002133   CNST   002
02  002136   ACON   20000724
02  002144   ACON   20000748
02  002152   ACON   20000516
02  002160   MVA    100606      000500    200516
02  002178   NTR    310014      002336
02  002190   CNST   0
02  002191   CNST   002
02  002194   ACON   20000772
02  002202   ACON   20000796
02  002210   ACON   20000516
02  002218   NTR    310002      002406
02  002230   CNST   0
02  002231   CNST   002
02  002234   BUN    27          000376
```

Figure D-26. BPLBND Example, Program Information Listing (Sheet 6 of 8)

```
                CODE LISTING FOR CODE BLOCK ICMBL2_ICM.PROG3, SEG #4

                    04  002560   NTR     310014      002432
                    04  002572   CNST    0
                    04  002573   CNST    000
                    04  002576   ACON    20000820
                    04  002584   ACON    20000844
                    04  002592   ACON    20002244
                    04  002600   NTR     310014      002432
                    04  002612   CNST    0
                    04  002613   CNST    000
                    04  002616   ACON    20000868
                    04  002624   ACON    20000892
                    04  002632   ACON    20002268
          04  002640    MVL     091200      202268    202268    202244
          04  002664    BUN     27          000376

        CODE LISTING FOR CODE BLOCK ICMBL4_ICM.OPENIT, SEG #3

          03  002432    BCT     300134
          03  002438    BUN     27          002454
          03  002446    ACON    001180
          03  002452    CNST    10
          03  002454    MVR     148266      404000    200916
          03  002472    BUN     27          000376


CODE LISTING FOR CODE BLOCK ICMBL4_ICM.WRITIT_proc, SEG #4

          04  002432    MVA     101212      F00020    200916
          04  002450    MVA     101212      F00028    200940
          04  002468    MVA     101212      F00036    200964
          04  002486    MVA     10A606      000916    001214
          04  002504    BCT     300234
          04  002516    BUN     27          002534
          04  002518    ACON    001180
          04  002524    CNST    000000
          04  002530    CNST    0200
          04  002534    MVR     14B266      404000    200916
          04  002552    BUN     27          000376
```

Figure D-26. BPLBND Example, Program Information Listing (Sheet 7 of 8)

```
        CODE LISTING FOR CODE BLOCK ICMBL4_ICM.CLOSIT, SEG #3

                    03  002480   BCT    300154
                    03  002486   BUN    27          002502
                    03  002494   ACON   001180
                    03  002500   CNST   40
                    03  002502   BUN    27          000376

        BINDER-SUPPLIED CODE:

                    01  000260   BCT    300214
                    01  000266   BUN    27          000282
                    01  000274   ACON   000508
                    01  000280   CNST   20
                    01  000282   MVA    100306      000508    000500
              01  000300    SUB    040606      000040    000500    000494
              01  000324    MVA    100606      000494    000040
              01  000342    NTR    310002      001852
              01  000354    CNST   000
              01  000358    BRT    3304FF      200008
              01  000370    BCT    300194
              01  000376    MVN    110303      000017    000118
              01  000394    NEQ    25          000410
              01  000402    EXT    32          F00000
              01  000410    MVA    100303      E00000    200480
              01  000428    MPY    05A203      320000    000118    000473
              01  000452    INC    01A505      001000    000473
              01  000470    EXT    32          300000
              01  000478    BUN    27          000000

INTER-SEGMENT TRANSFER CODE;

              01  001852    MVA    10B303      001896    200480
              01  001870    MVA    10A303      002000    000118
              01  001888    BUN    27          300164
              02  002292    MVA    108303      002432    200480
              02  002310    MVA    10A303      003000    000118
              02  002328    BUN    27          300196
              02  002336    MVA    10B303      002432    200480
              02  002354    MVA    10A303      004000    000118
              02  002372    BUN    27          300228
              02  002380    MVA    10B303      002560    200480
              02  002398    BUN    27          002354
              02  002406    MVA    10B303      002480    200480
              02  002424    BUN    27          002310

MAX ERROR LEVEL:     5
# OF WARNINGS:       1
# OF FATAL ERRORS:   0
BIND COMPLETED
```

Figure D-26. BPLBND Example, Program Information Listing (Sheet 8 of 8)

The next output is from PRINTANALYSIS. Since no blocks were specified in the statement, all code and data blocks in the program are listed. Following each are the names of code blocks which make reference to it.

The code requested by PRINTCODE is given next. Segment numbers and instruction addresses precede the lines of generated code for each procedure.

The BPLBND summary lines at the end of the listing are always produced. They give the maximum level of errors or warnings encountered, the number of warnings, the number of fatal errors, and an indication of whether a code file was produced.

APPENDIX E
COMMON BPL PROGRAMMING ERRORS

The following have been found to be common mistakes.

Forgetting that index registers are altered during the execution of the program. For example, IX1 has a negative value after a CASE statement is entered.

Forgetting that an indirect field length of 00 means that 100 will be used.

Forgetting that address controllers are included in the ADDRESS OF construct. For example if A is declared ALPHA at location 2000, then B := [A]; will result in 202000 being moved to B.

Forgetting that index registers add to the digit address, not the character address. For example in stepping through a character field with an index register, the register must be incremented by 2, not 1.

Name parameters cannot have their addresses altered in a meaningful way. Since they are referred to indirectly through the stack, adding a value to the address (as A.+2) will add to the stack address, rather than the resultant address.

Forgetting the "#" on a define, or the ")" on a parametric define call.

Defining key words.

Unmatched BEGIN/ENDS.

Having patch records out of sequence.

Omitting "&" on a line before the comments.

Leaving off the ",".

Including unmatched literal delimiters: "..... , @FIC.... , %C3D....

Boundary alignment problems; IA and ALPHA must be mod 2.

Formal and actual parameters must agree in procedure to get expected results.

When overriding a SIGNED INTEGER with a UN controller, a .+1 must be included to get past the sign.

Failing to declare something before it is used.

For example:

```
        GO TO L;
    L:
```

without having LABEL L; beforehand.

Not realizing that the error pointer may in some instances be pointing to a line previous to the one immediately above it.

APPENDIX F
EBCDIC, USASCII, AND BCL REFERENCE TABLE

GENERAL

This table reflects the internal EBCDIC structure in its sequential code arrangement for V Series Systems, plus the USASCII and BCL magnetic tape coding structures.

The two methods of creating the two-character codes are as follows:

1. 8-bit (byte) character code:

Decimal Equivalent			Binary		Decimal Equivalent	
	0		0000 0000		0	
	1		0001 0001		1	
	2		0010 0010		2	
Converted	3		0011 0011		3	Converted
	9		1001 1001		9	
A	{ 10		{ 1010 1010 }		10 }	A
B	{ 11		{ 1011 1011 }		11 }	B
C	{ 12	Undigit	{ 1100 1100 }	Undigit	12 }	C
D	{ 13	Character	{ 1101 1101 }	Character	13 }	D
E	{ 14		{ 1110 1110 }		14 }	E
F	{ 15		{ 1111 1111 }		15 }	F

Example:

If a memory dump reflects A3, the internal code would be 1010 0011. The highest sequential code is FF and the internal code is 1111 1111.

2. 6-bit (byte) character code:

Decimal	Binary	Decimal	
0	00 0000	0	
1	01 0100	4	Converted
2	10 1001	9	
3	11 1010 }	10 }	A
0	00 1100 }	12 }	C
2	10 1111 }	15 }	F

NOTE

There are only 64 unique 6-bit BCL tape codes. Therefore, where no BCL tape code is indicated in the table, there will be no BCL graphic character.

EBCDIC			USASCII	BCL		
8-Bit Internal Code	**Graphic**	**Card Code**	**8-Bit Internal Code**	**6-Bit Tape Code**	**Graphic**	**Card Code**
00	NULL	12-0-9-8-1	80			
01	SOH	12-9-1	81			
02	STX	12-9-2	82			
03	ETX	12-9-3	83			
04		12-9-4	84			
05	HT	12-9-5	85			
06		12-9-6	86			
07	DEL	12-9-7	87			
08		12-9-8	88			
09		12-9-8-1	89			
0A		12-9-8-2	8A			
0B	VT	12-9-8-3	8B			
0C	FF	12-9-8-4	8C			
0D	CR	12-9-8-5	8D			
0E	SO	12-9-8-6	8E			
0F	SI	12-9-8-7	8F			
10	DLE	12-11-9-8-1	90			
11	DC1	11-9-1	91			
12	DC2	11-9-2	92			
13	DC3	11-9-3	93			
14		11-9-4	94			
15	NL	11-9-5	95			
16	BS	11-9-6	96			
17		11-9-7	97			
18	CAN	11-9-8	98			
19	EM	11-9-8-1	99			
1A		11-9-8-2	9A			
1B		11-9-8-3	9B			
1C	FS	11-9-8-4	9C			
1D	GS	11-9-8-5	9D			
1E	RS	11-9-8-6	9E			
1F	US	11-9-8-7	9F			
20		11-0-9-8-1				
21		0-9-1				
22		0-9-2				
23		0-9-3				
24		0-9-4				
25	LF	0-9-5				
26	ETB	0-9-6				
27	ESC	0-9-7				
28		0-9-8				
29		0-9-8-1				
2A		0-9-8-2				
2B		0-9-8-3				
2C		0-9-8-4				

EBCDIC			USASCII	BCL		
8-Bit Internal Code	**Graphic**	**Card Code**	**8-Bit Internal Code**	**6-Bit Tape Code**	**Graphic**	**Card Code**
2D	ENQ	0-9-8-5				
2E	ACK	0-9-8-6				
2F	BEL	0-9-8-7				
30		12-11-0-9-8-1				
31		9-1				
32	SYN	9-2				
33		9-3				
34		9-4				
35		9-5				
36		9-6				
37	EOT	9-7				
38		9-8				
39		9-8-1				
3A		9-8-2				
3B		9-8-3				
3C	DC4	9-8-4				
3D	NAK	9-8-5				
3E		9-8-6				
3F	SUB	9-8-7				
40	SPACE		AO	10		
41		12-0-9-1				
42		12-0-9-2				
43		12-0-9-3				
44		12-0-9-4				
45		12-0-9-5				
46		12-0-9-6				
47		12-0-9-7				
48		12-0-9-8				
49		12-8-1				
4A	L	12-8-2	DB	3C		12-8-4
4B	.	12-8-3	AE	3B		
4C	<	12-8-4	BC	3E		12-8-6
4D	(	12-8-5	A8	3D		
4E	+	12-8-6	AB	3A		12-0
4F	X!	12-8-7	DE	3F		
50	&	12	A6	30		
51		12-11-0-1				
52		12-11-9-2				
53		12-11-9-3				
54		12-11-9-4				
55		12-11-9-5				
56		12-11-9-6				
57		12-11-9-7				
58		12-11-9-8				
59		11-8-1				

EBCDIC			USASCII	BCL		
8-Bit Internal Code	**Graphic**	**Card Code**	**8-Bit Internal Code**	**6-Bit Tape Code**	**Graphic**	**Card Code**
5A	]	11-8-2	DD	1E		0-8-6
5B	$	11-8-3	A4	2B		
5C	*	11-8-4	AA	2C		
5D	)	11-8-5	A9	2D		
5E	;	11-8-6	BB	2E		
5F	^	11-8-7	DC	2F	≤	
60	-	11	AD	20		
61	/	0-1		11		
62		11-0-9-2				
63		11-0-9-3				
64		11-0-9-4				
65		11-0-9-5				
66		11-0-9-6				
67		11-0-9-7				
68		11-0-9-8				
69		0-8-1				
6A			12-11			
6B	,	0-8-3	AC	1B		
6C	%	0-8-4	A5	1C		
6D	–	0-8-5	DF	1A	^=	0-8-2
6E	>	0-8-6	BE	0E		8-6
6F	?	0-8-7	BF	00		
70		12-11-0				
71		12-11-0-9-1				
72		12-11-0-9-2				
73		12-11-0-9-3				
74		12-11-0-9-4				
75		12-11-0-9-5				
76		12-11-0-9-6				
77		12-11-0-9-7				
78		12-11-0-9-8				
79		8-1				
7A	:	8-2	BA	OA		8-5
7B	#	8-3	A3	OB		
7C	@	8-4	CO	OC		
7D	’	8-5	A7	OF	≥	0-8-7
7E	=	8-6	BD	1D		0-8-5
7F	″	8-7	A2	1F		0-8-7
80		12-0-8-1				
81	a	12-0-1				
82	b	12-0-2				
83	c	12-0-3				
84	d	12-0-4				
85	e	12-0-5				
86	f	12-0-6				

EBCDIC			USASCII	BCL		
8-Bit Internal Code	**Graphic**	**Card Code**	**8-Bit Internal Code**	**6-Bit Tape Code**	**Graphic**	**Card Code**
87	g	12-0-7				
88	h	12-0-8				
89	i	12-0-9				
8A		12-0-8-2				
8B		12-0-8-3				
8C		12-0-8-4				
8D		12-0-8-5				
8E		12-0-8-6				
8F		12-0-8-6				
90		12-11-8-1				
91	j	12-11-1				
92	k	12-11-2				
93	l	12-11-3				
94	m	12-11-4				
95	n	12-11-5				
96	o	12-11-6				
97	p	12-11-7				
98	q	12-11-8				
99	r	12-11-9				
9A		12-11-8-2				
9B		12-11-8-3				
9C		12-11-8-4				
9D		12-11-8-5				
9E		12-11-8-6				
9F		12-11-8-7				
A0		11-0-8-1				
A1		11-0-1				
A2	s	11-0-2				
A3	t	11-0-3				
A4	u	11-0-4				
A5	v	11-0-5				
A6	w	11-0-6				
A7	x	11-0-7				
A8	y	11-0-8				
A9	z	11-0-9				
AA		11-0-8-2				
AB		11-0-8-3				
AC		11-0-8-4				
AD		11-0-8-5				
AE		11-0-8-6				
AF		11-0-8-7				
B0		12-11-0-8-1				
B1		12-11-0-1				
B2		12-11-0-2				
B3		12-11-0-3				

EBCDIC			USASCII	BCL		
8-Bit Internal Code	**Graphic**	**Card Code**	**8-Bit Internal Code**	**6-Bit Tape Code**	**Graphic**	**Card Code**
B4		12-11-0-4				
B5		12-11-0-5				
B6		12-11-0-6				
B7		12-11-0-7				
B8		12-11-0-8				
B9		12-11-0-9				
BA		12-11-0-8-2				
BB		12-11-0-8-3				
BC		12-11-0-8-4				
BD		12-11-0-8-5				
BE		12-11-0-8-6				
BF		12-11-0-8-7				
C0	(+)PZ	12-0				
C1	A	12-1	C1	31		
C2	B	12-2	C2	32		
C3	C	12-3	C3	33		
C4	D	12-4	C4	34		
C5	E	12-5	C5	35		
C6	F	12-6	C6	36		
C7	G	12-7	C7	37		
C8	H	12-8	C8	38		
C9	I	12-9	C9	39		
CA		12-0-9-8-2				
CB		12-0-9-8-3				
CC		12-0-9-8-4				
CD		12-0-9-8-5				
CE		12-0-9-8-6				
CF		12-0-9-8-7				
D0	(!)MZ	11-0	A1	2A	x	11-0
D1	J	11-1	CA	21		
D2	K	11-2	CB	22		
D3	L	11-3	CC	23		
D4	M	11-4	CD	24		
D5	N	11-5	CE	25		
D6	O	11-6	CF	26		
D7	P	11-7	D0	27		
D8	Q	11-8	D1	28		
D9	R	11-9	D2	29		
DA		12-11-9-8-2				
DB		12-11-9-8-3				
DC		12-11-9-8-4				
DD		12-11-9-8-5				
DE		12-11-9-8-6				
DF		12-11-9-8-7				

EBCDIC			USASCII	BCL		
8-Bit Internal Code	**Graphic**	**Card Code**	**8-Bit Internal Code**	**6-Bit Tape Code**	**Graphic**	**Card Code**
E0		0-8-2				
E1		11-0-9-1				
E2	S	0-2	D3	12		
E3	T	0-3	D4	13		
E4	U	0-4	D5	14		
E5	V	0-5	D6	15		
E6	W	0-6	D7	16		
E7	X	0-7	D8	17		
E8	Y	0-8	D9	18		
E9	Z	0-9	DA	19		
EA		11-0-9-8-2				
EB		11-0-9-8-3				
EC		11-0-9-8-4				
ED		11-0-9-8-5				
EE		11-0-9-8-6				
EF		11-0-9-8-7				
F0	0	0	B0	0A		
F1	1	1	B1	01		
F2	2	2	B2	02		
F3	3	3	B3	03		
F4	4	4	B4	04		
F5	5	5	B5	05		
F6	6	6	B6	06		
F7	7	7	B7	07		
F8	8	8	B8	08		
F9	9	9	B9	09		
FA		12-11-0-9-8-2				
FB		12-11-0-9-8-3				
FC		12-11-0-9-8-4				
FD		12-11-0-9-8-5				
FE		12-11-0-9-8-6				
FF		12-11-0-9-8-7				

APPENDIX G
BPL68

The following DEFINES constitute the BPL68 library file invoked by some internal Unisys software programs. The use of these items is neither supported nor encouraged by Unisys Corporation. This information is supplied solely for those with the need and the authorization to compile or modify programs which invoke BPL68.

```
DEFINE  IN_ =                   BEGIN BEGIN UNSEGMENTED #,
        OUT_ =                  END ELSE BEGIN UNSEGMENTED #,
        ELSE_ =                 END ELSE BEGIN UNSEGMENTED #,
        ELIF_ =                 END ELSE IF #,
        THEN_ =                 THEN BEGIN UNSEGMENTED #,
        UNTIL_(C) =             ; IF C THEN EXITLOOP; #,
        WHILE_(C) =             ; IF NOT C THEN EXITLOOP; #,
        END_ =                  END #,
        FI_ =                   FI #,
        OD_ =                   OD #,
        ESAC_ =                 ESAC #,
        EXITBLOCK_ =            EXITBLOCK #,
        EXITCOND_ =             EXITCOND #,
        EXITLOOP_ =             EXITLOOP #,
        EXITCASE_ =             EXITCASE #,
        GOTOTOPLOOP_ =          TOPLOOP #,
        TOPLOOP_ =              TOPLOOP #;
```

INDEX

Special Characters

&, signifying a comment 5-29
:=, definition of 5-7

A

ACCEPT data communications statement,
  definition of 7-2
  syntax of 7-2
ACCEPT statement,
  definition of 5-2
  syntax of 5-2
accessing data areas, with multiple ICMs 4-6
accumulator arithmetic commands 5-3
accumulator constructs,
  definition of 5-3
accumulator instructions,
  DACCUM 5-3
  examples of 5-4
  IACCUM 5-3
  RACCUM 5-3
  when generated 5-3
accumulator load commands 5-3
accumulator manipulate instruction,
  functions of 5-3
accumulator store commands 5-3
ACTION 0 statement, for reader/sorters,
  definition of 9-2
  syntax of 9-2
ACTION 4 statement, for reader/sorters,
  definition of 9-3
  syntax of 9-3
ACTION 6 statement, for reader/sorters,
  definition of 9-4
  syntax of 9-4
ACTION 8 statement, for reader/sorters,
  definition of 9-5
  syntax of 9-5
ACTION 10 statement, for reader/sorters,
  definition of 10-2
  syntax of 10-2
ACTION 11 statement, for reader/sorters,
  definition of 10-3
  syntax of 10-3
ACTION 12 statement, for reader/sorters,
  definition of 10-4
  syntax of 10-4
ACTION 13 statement, for reader/sorters,
  definition of 10-5
  syntax of 10-5
ACTION 14, for reader/sorters,
  definition of 10-6
  syntax of 10-6
ACTION 15, for reader/sorters,
  definition of 10-7
  syntax of 10-7
ADDRESS declaration statement,
  definition of 4-2
  example of 4-2, 4-3
  syntax of 4-2
address errors, compile-time, reasons for 11-6
address, storing in a field 4-10
addressing, and the DYNAMIC declaration statement 4-19
alignment
  control 3-4
  of alphabetic data 3-4
  of numeric data 3-4
alpha types, using in an arithmetic operation 4-9
alphabetic data moves,
  and alignment 3-4
alphabetic data types,
  and arrays 4-10
  and justification 4-12
  and modulo declaration 4-11
  and use of DYNAMIC 4-19
ampersand, signifying a comment 5-29
areas,
  assigning by cylinder boundary 4-25
  default number for a file 4-23
  specifying number of for a file 4-23
  specifying number of records in 4-24
arithmetic expressions,
  compound 5-18
  examples of 5-18
arithmetic operations,
  assignment overrides 5-11
  fixed length 5-3
arithmetic operators,
  definition of 2-2
  list of 5-10
  use of 2-2
ARM statement, definition of 5-5
ARMed programs, how to disarm 5-32
arrays,
  accessing elements of 4-10
  and subscripting 2-7
  and subscripting 4-10
  and various data types 4-10
  defining 4-10

definition of 2-7
example of 2-8
function of 2-7
initializing contents of 4-12
presetting contents of 4-12
assignment
overrides,
and the assignment statement 5-8
example of 5-9
in arithmetic operations 5-11
list of 5-8
assignment
statement,
compute option 5-10
definition of 1-1, 2-5, 3-4, 5-7
definition of 5-7
example of 5-9
exchange option 5-9
logical operators 5-12
move data option 5-10
SEGDICT option 5-13
SEGMENT option 5-14
special BCT option 5-13
assignment
symbol,
definition of 2-2
use of 2-2
at sign, use of 2-10
attributes,
for port files,
list of 8-15
for ports and subports,
definition of 8-1, 8-14
establishing 8-6
obtaining 8-3
setting 8-9
attributes,
subport files,
list of 8-16

B

BASE special identifier, description of 2-6
basic symbols,
definition of 2-2
use of 2-2
batch counter,
in reader sorters, incrementing 9-4
BCL character set,
table of F-1
BCT, *see* Branch Communicates
BIT declaration,
definition of 4-4
example of 4-4
syntax of 4-4
bit
reset 4-7
set 4-7
types 4-9
bit,
declaring as data 4-9
manipulating or checking 5-12
block format 2-13
blocks,
and addressing 4-19
as segments 2-14
definition of 1-1
definition of 2-13
definition of B-1
disjoint 2-13
exiting 5-43
handling duplicate labels in 4-31
internal structure of 2-13
nested 2-13
specifying number of records in 4-24
BNA network software,
requirements for 8-1
Boolean operators,
and the assignment statement 5-12
BPL
character set 2-2
compiler input options 11-2
compiler,
and CADE Editor files 11-8
and multiprogramming environments 11-7
directing statements 3-4
examples of compilations 11-6
function of 1-1
operating 11-1
warning and error messages C-1
key words,
list of A-1
language characteristics 2-1
programs,
common mistakes when writing E-1
entry point defined 2-14
example of structure B-2
form of B-1
format of 2-13
hints on how to write B-1
size considerations 2-14
reserved words 2-4
list of A-1
source program,
definition of 1-1
statements,

definition of 3-1
syntax notations 2-1
syntax,
arithmetic operators 2-2
arrays 2-7
assignment symbol 2-2
at sign 2-10
basic symbols 2-2
block format 2-13
blocks 2-13
braces 2-1
brackets 2-2
conditional relations 2-3
consecutive periods 2-2
controller fields 2-11
duplicate identifiers 2-6
ellipsis 2-2
format of programs 2-13
identifiers 2-5
key words 2-1
language statements 2-5
literals 2-10
examples of 2-10
logical operators 2-2
lower case words 2-1
non-numeric literals 2-10
numeric literals 2-10
optional words 2-1
period 2-2, 2-11
punctuation 2-4
quotation marks 2-10
scope of identifiers 2-6
separators 3-1
special identifiers 2-6
statements 3-1
subscripts 2-7
undigit numeric literals 2-10
BPL,
advantages of 1-1
and blocks 1-1
and data communications 7-1
and Independently Compiled Modules D-1
and program size 2-14
assignment statements for 1-1
compound statements for 1-1
declarations for 1-1
example of program structure B-2
executable statements for 1-1
form of programs in B-1
format of 1-1
hints on writing programs in B-1
uses for 1-1
BPLBND program binder,
and BINDER D-27
definition of D-27
use of D-27
BPLBND statements,
END,
definition of D-46
syntax of D-46
FATAL,
definition of D-32
syntax of D-32
NOEXTEND,
definition of D-33
syntax of D-33
OPTIONAL,
definition of D-31
syntax of D-31
OVERLAY,
definition of D-44
syntax of D-44
PRINT,
definition of D-34
syntax of D-34
PRINTALL,
definition of D-35
syntax of D-35
PRINTANALYSIS,
definition of D-36
syntax of D-36
PRINTCODE,
definition of D-37
syntax of D-37
PRINTSEGANALYSIS,
definition of D-38
syntax of D-38
PROGRAMLIMIT,
definition of D-39
syntax of D-39
PROGRAMSIZE,
definition of D-40
syntax of D-40
REQUIRED,
definition of D-30
syntax of D-30
SEGMENT,
definition of D-42
syntax of D-42
STACKSIZE,
definition of D-41
syntax of D-41
BPLBND usage examples D-56
braces,
definition of 2-1
use of 2-1

brackets,
definition of 2-2
use of 2-2
Branch Communicates,
indexing of identifiers 5-18
special,
and assignment statement 5-13
list of 5-13
breakout,
and restart, specifying 5-21
and resulting program actions 5-5
definition of 4-26
directing output 4-8
specifying 4-8
specifying for a sort 5-71
breakout result descriptors, description of 5-6
BREAKOUT statement,
definition of 5-21
syntax of 5-21
buffer access method,
and NO WORKAREA 4-24
buffers,
accessing records from 4-24
maximum possible 4-24
specifying number of 4-24
stream mode 7-14

C

calling procedures 3-1
considerations for 3-1
CANCEL data communications statement,
definition of 7-3
syntax of 7-3
cancelling IO descriptors 7-3
CANDE Editor files,
compiling 11-8
CASE statement,
and compile-time address errors 11-6
and efficiency of code 5-23
as related to the IF statement 3-4
definition of 3-4
definition of 5-22
examples of 5-24
exiting 5-44
syntax of 5-22
CDATE declaration statement,
definition of 4-5
syntax of 4-5
character count,
stored during a scan 5-58
character set 2-2
characters,
searching for specific 5-58
Class I, reserved words defined 2-4
Class II, reserved words defined 2-4
CLOSE statement,
definition of 5-25
for port files 8-2
syntax of 5-25, 8-2
for reader/sorters,
definition of 10-8
syntax of 10-8
operations caused by 5-26
code translation,
declaring option for 4-26
colon-equal (:=),
definition of 5-7
COMMENT statement,
definition of 5-29
syntax of 5-29
COMMON declaration statement,
definition of 4-6
examples of 4-6
syntax of 4-6
COMMON declarations,
restrictions on 4-6
COMPARE statement,
definition of 5-30
differences from the IF statement 5-30
syntax of 5-30
compilation,
and CANDE Editor files 11-8
examples 11-6
memory space for 11-6
compile-time address errors,
reasons for 11-6
compile-time errors,
and the DEFINE statement 4-16
compiler directing statements 6-1, 6-2
definition of 2-5, 3-4
ICM 6-6
IFF 6-7
LIBR 6-4
PAGE 6-5
compiler,
flags,
setting and resetting 6-2
function of 1-1
input options 11-2
operation, instructions for 11-1
options, list of 11-2
warning and error messages C-1
compiling,
BPL programs 11-1
conditional 6-2

in multiprogramming environments 11-7
compound arithmetic expressions,
see assignment statement
compound expressions,
and the IF statement 5-50
compound statements,
definition of 1-1
CONDCANCEL data communications statement,
definition of 7-4
syntax of 7-4
conditional compiling,
definition of 6-2
example of 6-3
syntax of 6-2
conditional expressions,
and CASE 3-4
and IF 3-3
compound 5-18
examples of 5-18
conditional relations,
definition of 2-3
use of 2-3
conditional statements,
definition of 2-5
exiting 5-45
consecutive periods,
definition of 2-2
use of 2-2
constants,
producing a string of 5-79
contention conditions,
how handled 4-29
CONTROL declaration statement,
and BREAKOUT option 4-8
and DICTIONARY option 4-8
and EXTENDED option 4-8
definition of 4-7
options 4-7
syntax of 4-7
control instructions, executing 5-95
CONTROL options 4-7
control points,
how they are established 4-26
specifying in a program 4-31
control statements,
definition of 2-5, 3-1
function of 3-1
control transfer,
and ENTER 5-41
and EXITBLOCK 5-43
and the GO statement 5-49
to an Independently Compiled Module 5-80
TOPLOOP statement 5-81
controller field overrides,
example of 2-13
list of 2-11
controller field reserved words 2-11
controller fields,
definition of 2-11
function of 2-11
overrides 2-11
COPY statement,
definition of 5-31
syntax of 5-31
core to core function,
compared to port files 8-1
difference from STOQUE 5-76
cylinder boundaries, and area assignment 4-25

D

DACCUM accumulator instruction 5-3
data areas,
defining 4-9
initial contents of 4-12
initializing 4-9
presetting in data declaration 4-12
Data Communications Module,
requirements for 7-1
data communications statements,
ACCEPT 7-2
CANCEL 7-3
CONDCANCEL 7-4
DISPLAY 7-5
ENABLE 7-6
FILL 7-7
INTERROGATE 7-9
READ 7-13
READY 7-14
TRANSTBL 7-15
WAIT 7-16
WRITE 7-17
WRITEREAD 7-18
WRITEREADTRANS 7-19
WRITETRANSREAD 7-20
data communications
and BPL 7-1
translate tables 7-15
data declaration statement,
and declaring arrays 4-10
definition of 4-9
examples of 4-13 thru 4-15
syntax of 4-9
data declaration,
and presetting data areas 4-12
and uninitialized identifiers 4-12

data translation,
algorithm for 5-83
example of 5-84, 5-87
data types,
alphabetic 4-9
in an arithmetic operation 4-9
bit 4-9
double 4-9
fixed double 4-9
fixed integer 4-10
fixed real 4-10
indirect 4-10
integer 4-10
numeric 4-10
real 4-10
signed integer 4-10
data,
bit declared as 4-9
displaying on the ODT 5-33
entering through an ODT 5-2, 7-2
moving and editing 5-40
passing from one program to another 5-48
passing from remote devices 7-18 thru 7-20
passing to remote devices 7-18 thru 7-20
printing on a remote SPO 7-5
transferring between processes 5-76
using translation tables on 5-83
DATACOMM
ACCEPT statement, definition of 7-2
CANCEL statement, definition of 7-3
CONDCANCEL statement, definition of 7-4
DISPLAY statement, definition of 7-5
ENABLE statement, definition of 7-6
FILL statement, definition of 7-7
INTERROGATE statement, definition of 7-9
READ statement, definition of 7-13
READY statement, definition of 7-14
TRANSTBL statement, definition of 7-15
WAIT statement, definition of 7-16
WRITE statement, definition of 7-17
WRITEREAD statement, definition of 7-18
WRITEREADTRANS statement, definition of 7-19
WRITETRANSREAD statement, definition of 7-20
declaration statements,
ADDRESS 4-2
CDATE 4-5
COMMON 4-6
CONTROL 4-7
DEFINE 4-16
definition of 2-5, 3-1, 4-1
definition of 3-1
definition of 4-1
DYNAMIC 4-19
FILE 4-21
function of 3-1
LABEL 4-31
list of 4-1
PICTURE 4-32
PROCEDURE 4-33
SUBROUTINE 4-37
UNSEGMENTED 4-38
DATA 4-9
declarations,
and routines 4-16
BIT 4-4
definition of 1-1
global 2-6
local 2-6
PORT 8-6
DEFINE declaration statement,
and nesting capabilities 4-16
and parameters 4-16
and reserved words 4-16
description of 4-16
examples of 4-17, 4-18
syntax of 4-16
destination field,
and assignment operations 3-4
DISARM statement,
definition of 5-32
see also ARM statement
syntax of 5-32
disjoint blocks, definition of 2-13
diskpacks,
limiting file assignment 4-23
DISPLAY data communications statement,
definition of 7-5
syntax of 7-5
DISPLAY statement,
definition of 5-33
syntax of 5-33
DO statement,
and EXITLOOP 5-46
definition of 5-34
examples of 5-35, 5-36
syntax of 5-34
DO statements,
and UNTIL 3-2
and WHILE 3-3
definition of 3-2
example of 3-2-3
documenting source programs 5-29
dollar options, list of 11-2
dollar sign,
and compiler directing statements 6-2
dot, *see* period

double precision data 4-9
and arrays 4-10
double precision operations 4-9
double types 4-9
DOZE statement,
definition of 5-34, 5-38
syntax of 5-34, 5-38
DO_ loops, and the TOPLOOP statement 5-81
DUMP statement,
definition of 5-39
examples of 5-39
syntax of 5-39
dumping memory data 5-39
duplicate identifiers,
use of 2-6
dynamic area,
defined 4-19
how to manage it 4-20
dynamic area, *see also* DYNAMIC declaration statement
DYNAMIC declaration statement,
and memory space 4-19
and modulo adjustment 4-19
and program space 4-19
description of 4-19
examples 4-20
syntax of 4-19

E

EBCDIC character set, table of F-1
EDIT statement,
definition of 5-40
examples of 5-40
syntax of 5-40
Editor files, compiling 11-8
ellipsis,
definition of 2-2
use of 2-2
ENABLE data communications statement,
and WAIT 7-6
definition of 7-6
syntax of 7-6
enquiry,
recognizing from a remote device 7-6
ENTER statement,
and importance of EXIT 5-41
definition of 5-41
syntax of 5-41
entry point, definition 2-14
equivalence tables,
used with data translation 5-83
error messages,
displaying on a remote SPO 7-5
how to read C-1
list of C-1
errors,
caused by insufficient memory allocation 11-6
common programming causes E-1
compile-time address, reasons for 11-6
hardware-detected 5-6
ESAC, use with CASE_ 5-23
ETX character,
as used with SPOMESSAGE 5-74
exceptions, data passed to program 5-5
executable statements,
ACCEPT 5-2
ARM 5-5
assignment 5-7
BREAKOUT 5-21
CASE 5-22
CLOSE 5-25
COMMENT 5-29
COMPARE 5-30
control 3-1
COPY 5-31
definition of 1-1, 2-5, 3-1
DISARM 5-32
DISPLAY 5-33
DO 5-34
DOZE 5-38
DUMP 5-39
EDIT 5-40
ENTER 5-41
EXIT 5-42
EXITBLOCK 5-43
EXITCASE 5-44
EXITCOND 5-45
EXITLOOP 5-46
EXITROUTINE 5-47
FILL 5-48
function of 3-1, 5-1
GO 5-49
IF 5-50
LOCK 5-54
OPEN 5-55
OVERLAY 5-57
procedure call 5-58
READ 5-59
SCAN 5-58
SEARCH 5-63
SEARCH DELINK 5-65
SEARCH LINK 5-65
SEEK 5-68
SORT 5-70
SORT RETURN 5-72
SPACE 5-73

SPOMESSAGE 5-74
STOP 5-75
STOQUE 5-76
STORE 5-79
subroutine call 5-80
TOPLOOP 5-81
TRACE 5-82
TRANSLATE 5-83
types of 3-1
UNLOCK 5-89
WAIT 5-90
WHILE 5-92
WRITE 5-93
ZIP 5-95
execution,
suspending or stopping 5-75, 5-90
EXIT statement,
and ENTER 5-42
definition of 5-42
syntax of 5-42
EXITBLOCK statement,
definition of 5-43
syntax of 5-43
EXITCASE statement,
definition of 5-42, 44
syntax of 5-42, 5-44
EXITCOND statement,
definition of 5-45
syntax of 5-45
EXITLOOP statement,
and DO statement, example of 5-37
definition of 5-46
syntax of 5-46
EXITROUTINE statement,
definition of 5-47
syntax of 5-47
expressions,
arithmetic 5-18
conditional 5-18
order of evaluation 5-18
order of precedence of 5-18
extended addressing feature 4-8
EXTERNAL declaration statement,
definition of D-18

F

FALSE,
and its value in a data area 4-12
FIB,
see File Information Block
field,
defining to store an address 4-10
justifying data in 4-12
FILE declaration statement,
description of 4-21
syntax of 4-21
to specify file size 4-23
File Information Block,
and the IX2 flag 4-24
file
labels,
specifying type of 4-29
names,
internal and external 11-7
size,
specifying number of records per block 4-24
types,
default 4-26
list of 4-26
files,
and parity checking 4-24
assigning to one or more diskpacks 4-23
avoiding duplicate names when multiprogramming 4-24
checking for presence of on disk 5-14
checking for presence of on disk pack 5-14
creating, and the OPEN statement 5-55
declaring an optional file 4-26
default size 4-23
initiating processing of 5-55
library, creating 6-4
programmatically closing 5-25
random,
accessing with SEEK 5-68
specifying 4-26
serial, specifying 4-26
shared, specifying 4-26
specifying file size 4-23
specifying records per area 4-24
status when using STOP 5-75
tape,
automatic purging 4-25
specifying life of 4-25
unlocking blocks of 5-89
FILL data communications statement,
definition of 7-7
FILL statement,
definition of 5-48
syntax of 5-48
see also DATACOMM FILL
fixed data types,
and modulo declaration 4-11
fixed double precision data 4-9
fixed double types 4-9
fixed integer data 4-10
and modulo declaration 4-11

fixed operands,
and mixing with integer arithmetics 4-8
fixed real data 4-10
and modulo declaration 4-11
format
of blocks 2-13
of BPL programs 2-13
program,
providing BPL information on 4-7
forms,
halting a program for special forms 4-26
forward procedure,
example of 4-35
see also procedures
function output parameters,
definition of 8-17
interrogating their values 8-18

G

GET statement,
for port files 8-3
examples of 8-3
syntax of 8-3
global declarations 2-6
GO statement,
definition of 5-49
syntax of 5-49

H

halting execution of a program 5-75
hardware
features,
providing BPL information on 4-7
names,
valid for BPL 4-23
reader/sorter,
activating through BPL 9-1, 10-1
types,
valid for BPL 4-23

I

I/O operations, cancelling 7-4
IACCUM accumulator instruction 5-3
ICM compiler directing statement,
definition of 6-6
example of 6-6
syntax of 6-6
ICM1s,
see Independently Compiled Modules, Type I
ICM2s,
see Independently Compiled Modules, Type II
ICM3s,
see Independently Compiled Modules, Type III
ICMs, see Independently Compiled Modules 5-80
identifiers,
controller field override 2-12
definition of 2-5-6
duplicate 2-6
function of 2-5
rules for 2-5
scope of 2-6
special 2-6
use of 2-5, 2-6
IF statement,
and compile-time address errors 11-6
and EXITCOND 5-45, 5-53
definition of 3-3, 5-50
differences from the COMPARE statement 5-30
example of 3-3, 5-53
for port files 8-4
examples of 8-4
syntax of 8-4
syntax of 5-50, 5-53
IFF compiler directing statement,
definition of 6-7
example of 6-7
syntax of 6-7
Independently Compiled Modules,
accessing data areas 4-6
creating Type I 6-6
EXTERNAL declaration for D-18
function of D-1
passing control to 5-80
Type I D-1, D-2
Type II D-1, D-13
binding D-27
Type III D-1, D-13
types of D-1
index registers,
and ADDRESS 4-2
and the File Information Block 4-24
cautions concerning 2-7
controller field override 2-12
example of 2-8
special identifiers for 2-6
storing addresses in 5-63, 5-65
use of 2-7
indirect data types 4-10
and modulo declaration 4-11
difference from integers 4-10
input files,
declaring as optional 4-26
input/output operations, cancelling 7-4

instruction sets,
default 4-7
specifying 4-7
integer
arithmetics, and mixing with fixed operands 4-8
data types 4-10
and arrays 4-10
and justification 4-12
and modulo declaration 4-11
and use of DYNAMIC 4-19
difference from indirect data types 4-10
interprogram communication,
definition of 8-1
INTERROGATE data communications statement,
definition of 7-9
IO descriptors, cancelling 7-3
IX1 special identifier,
description of 2-6
IX2 special identifier,
description of 2-6
IX3 special identifier,
description of 2-6

J

JOBINFO, allocating space for 5-19
justification of data in fields 4-12
justifying data in fields 4-12

K

key words,
definition of 2-1, A-1
list of A-1
use of 2-1
keyboard commands,
valid through SPOMESSAGE 5-74
passing to the MCP 5-74

L

LABEL declaration statement,
description of 4-31
syntax of 4-31
label routines,
definition of 4-29
list of 4-29
labels,
and scope considerations for 2-6
file, specifying type of 4-29
language statements,
definition of 2-5
types of 2-5
LIBR compiler directing statement,
LIBR compiler directing statement,
definition of 6-4
examples of 6-4
libraries, as procedures 5-31
library files, creating 6-4
library routines,
including in program 5-31
LINKAGE construct, definition of D-8
literals,
definition of 2-10
examples of 2-10
function of 2-10
non-numeric 2-10
numeric 2-10
undigit numeric 2-10
local
declarations 2-6
variables,
and low-level procedures 4-34
pertaining to procedures 4-34
local-only ports,
definition of 8-1
LOCK statement,
and SEEK 5-54
definition of 5-54
see also UNLOCK 5-89
syntax of 5-54
use of WRITE after 5-93
locked files,
and considerations for future writes 5-54
locking a file,
with the OPEN statement 5-55
logical operators,
and the assignment statement 5-12
definition of 2-2, 5-52
example of 5-12
list of 5-52
use of 2-2
logical records,
releasing for output 5-93
looping, with the DO statement 5-34
loops, exiting 5-46
lower case words,
definition of 2-1
use of 2-1

M

Magnetic Ink Character Recognition,
and reader/sorters 9-1, 10-1
as a file type 4-27
control options for 4-27

MCP, passing messages to 5-74
memory
  and overlays,
    and the OVERLAY statement 5-57
  buffer,
    transferring data to and from 5-76
  data,
    dumping to printer or disk 5-39
  division of areas D-44
  locations,
    passing data from to remote devices 7-18 thru 7-20
    passing data to from remote devices 7-18 thru 7-20
  management,
    special identifiers for 2-6
  size,
    determining 4-20
    specifying 4-7
  space,
    allocation for a sort 5-70
    and requirements for compilation 11-6
    and use of DYNAMIC 4-19
    computing for SORT intrinsic 4-28
    increasing, see CONTROL 4-19
    required for buffers 4-24
messages, passing to the MCP 5-74
MICR,
  see Magnetic Ink Character Recognition
microfilm,
  advancing on reader/sorters 10-6
  generating image count marks on 10-3
mix functions, list of 5-17
MIXTBL BCT, generating 5-17
MIXTBL information,
  providing space for 5-19
MOD declaration,
  definition of 4-11
  see also modulo
modulo,
  and use of DYNAMIC 4-19
  declaration,
    and various data types 4-11
    defaults for 4-11
    definition of 4-11
  significance for translation tables 5-83
move links, definition of 5-9
multiprogramming environments,
  compiling in 11-7

N

nested blocks, definition of 2-13
nesting, and DEFINE 4-16
non-numeric literals,
  definition of 2-10
  function of 2-10
nonlocal,
  see global
numeric data moves,
  and alignment 3-4
numeric data types 4-10
  and justification 4-12
  and modulo declaration 4-11
numeric literals,
  definition of 2-10
  function of 2-10

O

OPEN statement,
  and tape files 5-55
  definition of 5-55
  for port files 8-5
    syntax of 8-5
  for reader/sorters,
    definition of 9-6, 10-9
    syntax of 9-6, 10-9
  syntax of 5-55
operating the BPL compiler 11-1
operators, arithmetic,
  list of 5-10
optional files, uses for 4-26
optional words,
  definition of 2-1
  use of 2-1
OVERLAY statement,
  definition of 5-57
  syntax of 5-57
overlays,
  and limitations on calling procedures 5-58
  declaring segmentation for D-44
  default levels 4-33
  segmenting procedures 4-33
  structuring 4-33
overriding segmentation 2-14

P

PAGE compiler directing statement,
  definition of 6-5
  syntax of 6-5
pages,
  default number for a file 4-23
  specifying number of for a file 4-23
parameters,
  function output 8-17
  specifying for a procedure 4-34

used in a DEFINE statement 4-16
value, definition of 4-34
PARAMETRIC DEFINE statement,
description of 4-16
parity checking,
when writing files 4-24
parity errors,
and the SORT statement 5-71
period,
and the ZIP statement 5-95
definition of 2-2
use of 2-2, 2-11
peripheral names, valid for BPL 4-23
peripherals,
list of recording modes for 4-25
specifying recording modes for 4-25
PICTURE declaration statement,
description of 4-32
example of 4-32
syntax of 4-32
pocket light, on reader sorters,
illuminating 9-3, 10-3
polling routines,
and the DOZE statement 5-38
PORT declaration,
definition of 8-6
examples of 8-7
syntax of 8-6
port file attributes,
definition of 8-14
list of 8-15
port file statements,
CLOSE 8-2
GET 8-3
IF 8-4
OPEN 8-5
READ 8-8
SET 8-9
WAIT 8-11
WRITE 8-13
port files,
accessing fields in FIB 8-17
and subports 8-1
declaration for 8-6
definition of 8-1
fields updated after IO 8-17
how to use them 8-1
local only 8-1
overrides on 8-18
remote 8-1
remote, and BNA 8-1
preterm error codes,
description of 5-6
printers,
forward spacing of paper 5-73
vertical positioning with WRITE 5-93
procedure call statement,
considerations for 3-1
definition of 3-1, 5-58
examples of 5-58
function of 3-1
procedure calls B-5
examples of 5-58
syntax for 5-58
PROCEDURE declaration statement,
description of 4-33
example of 4-35
structure of a procedure 4-34
syntax of 4-33
procedure statements,
definition of 2-5
procedures,
as library data 5-31
calling 3-1, 5-58, B-5
calling before they are declared 4-33
calling recursively 4-34
calling, considerations for 3-1
definition of B-1
differences from routines 4-28
example of 4-35
forward, definition of 4-33
invoking 4-34
low level, defined 4-34
restrictions on 4-34
when they are executed 4-33
program,
definition of 1-1
determining environment of 5-15
execution,
resuming after a STOP 5-75
suspending 5-90, 8-11
tracing of 5-82
format,
providing BPL information on 4-7
parameter branch communicates,
use of 5-15
size,
and use of DYNAMIC 4-19
maximums 4-8
specifying 4-7, 4-8
specifying the segment dictionary start 4-8
stack, and addressing 4-19
suspension,
DOZE statement 5-38
with DATACOMM WAIT 7-16
terminations, causing graceful 5-5

programming,
common errors when E-1
hints on writing programs B-1
programs,
arming to handle interrupts 5-5
common mistakes when writing E-1
establishing restart capability 4-26
form of B-1
hints on how to write B-1
including library routines 5-31
passing data between programs 5-48
size considerations 2-14
terminating execution of 5-75
punctuation in BPL 2-4

Q

quotation marks, use of 2-10

R

RACCUM accumulator instruction 5-3
random files,
accessing with SEEK 5-68
specifying 4-26
read constructs, list of 5-60
READ data communications statement,
definition of 7-13
syntax of 7-13
READ statement,
definition of 5-59
for port files 8-8
and OPEN 8-8
examples of 8-8
syntax of 8-8
for reader/sorters,
definition of 9-7, 10-10
syntax of 9-7, 10-10
syntax of 5-59
reader/sorter
characteristics, obtaining 10-5
file functions, performing 9-1, 10-1
files, closing 10-8
hardware,
and use routines 9-1, 10-1
activating through BPL 9-1, 10-1
statements,
ACTION 0 9-2
ACTION 4 9-3
ACTION 6 9-4
ACTION 8 9-5
ACTION 10 10-2
ACTION 11 10-3
ACTION 12 10-4
ACTION 13 10-5
ACTION 14 10-6
ACTION 15 10-7
CLOSE 10-8
OPEN 9-6, 10-9
READ 9-7, 10-10
status, obtaining 10-4
reader/sorters,
initiating processing 9-6, 10-9
advancing microfilm on 10-6
and generating count marks on microfilm 10-3
and microfilm 10-3
delaying start of read 9-5
illuminating pocket light 9-3, 10-3
incrementing batch counter 9-4
initiating flow feed 10-7
reading records from 9-7, 10-10
READY data communications statement,
definition of 7-13, 7-14
syntax of 7-14
real data types 4-10
and single precision operations 4-10
real numbers,
and arrays 4-10
defined for double precision 4-9
recording modes,
list of allowable 4-25
specifying for a peripheral 4-25
records,
accessing for a READ or WRITE 5-68
blocked and unblocked 4-24
forward or reverse spacing 5-73
moving to the program's work area 5-59
reader/sorter, reading 9-7, 10-10
reader/sorter, reading 9-7
specifying names for 4-24
specifying number per area 4-24
specifying number per block 4-24
specifying size of 4-24
specifying variable-length 4-25
recursive procedures,
definition of 4-34
relational operators, list of 5-51
remote devices,
obtaining result descriptors for 7-9
passing data to and from 7-18 thru 7-20
recognizing enquiry from 7-6
stopping the flow of data from 7-3
stopping the flow of data to 7-3
remote ports,
and BNA 8-1
definition of 8-1

remote SPO,
printing data on 7-5
receiving data from 7-2
repetitive execution, DO statement 5-34
rerunning a program,
how to set up files for 4-26
reserved words,
and the DEFINE statement 4-16
class I defined 2-4
class II defined 2-4
definition of 2-4, A-1
list of A-1
resource utilization,
efficient use of 4-9
result descriptor digits,
list of 7-10
status of 7-11
result descriptors,
classes of 5-6
obtaining for remote devices 7-9
rewinding a file,
and the OPEN statement 5-56
ringing signal, recognizing 7-6
routines,
allowable types 4-28
declaring with DEFINE 4-16
definition of 4-28-29
differences from procedures 4-28
exiting 5-47
functions of 4-28
label, definition of 4-29
when they are entered 4-29
running the BPL compiler 11-1

S

SCAN statement,
definition of 5-58
syntax of 5-58
scope of identifiers 2-6
and duplicate identifiers 2-6
scope,
and duplicate label names 4-31
and use of identifiers 2-6
as applies to labels 2-6
SEARCH DELINK statement,
definition of 5-65
syntax of 5-65
SEARCH LINK statement,
definition of 5-65
syntax of 5-65
search linked list instruction 4-7
SEARCH statement,
definition of 5-63
searching for a specific character,
the SCAN statement 5-58
seek constructs, list of 5-69
SEEK statement,
definition of 5-68
syntax of 5-68
segment dictionary entries,
accessing 5-13
segment dictionary, specifying 4-8
segmentation 2-14
segmentation overrides 2-14
segmentation, and blocks 2-14
segments,
calling overlayable 5-57
making variables segment relative 4-9
separators, list of 3-1
serial files, specifying 4-26
SET statement, for port files 8-9
examples of 8-10
syntax of 8-9
shared disk seek constructs,
list of 5-69
shared files,
locking blocks of 5-54
specifying 4-26
signed identifier,
controller field override 2-12
signed integer data types 4-10
signed integers,
and arrays 4-10
signed numeric literal,
controller field override 2-11
single precision operations,
and real numbers 4-10
size considerations, BPL programs 2-14
soft interrupt routines,
definition of 5-5
soft interrupt toggle,
and ARM statement 5-6
soft interrupts, enabling 5-5
SORT files,
and the CLOSE statement 5-25
SORT intrinsic,
computing minimum memory for 4-28
SORT RETURN statement,
definition of 5-72
syntax of 5-72
SORT statement,
definition of 5-70
syntax of 5-70
SORT, intrinsic, invoking 5-70
sorter/reader hardware,

activating through BPL 9-1, 10-1
source
field, and assignment operations 3-4
program, definition of 1-1
SPACE statement,
definition of 5-73
syntax of 5-73
special identifiers, definition of 2-6
SPOMESSAGE statement,
definition of 5-74
syntax of 5-74
valid keyboard commands for 5-74
stack
addresses, and segment size 4-8
pointers, readjusting,
warning about 4-20
relative variables 4-9
size, controlling 4-7
stack,
copying control information to 5-41
copying parameters to 5-41
stalemate conditions,
how handled 4-29
statements,
assignment 2-5, 3-4
binding together 3-2
CASE 3-4
compiler directing 2-5, 3-4
conditional 2-5
control 2-5, 3-1
declaration 2-5, 3-1, 4-1
definition of 3-1 thru 3-4
DO 3-2-3
executable 2-5, 3-1
types of 3-1
how to terminate 3-1
IF 3-3
PARAMETRIC DEFINE 4-16
procedure 2-5
procedure call 3-1
types of 3-1
UNTIL 3-2
WHILE 3-3
STOP statement,
definition of 5-75
resuming execution after using 5-75
syntax of 5-75
STOQUE module,
definition of 5-76
STOQUE parameter block,
format of 5-76
STOQUE statements, syntax of 5-76
STOQUE type statements,
definition of 5-76
STOQUE,
difference from core to core 5-76
see also storage queue
storage queue parameter block,
format of 5-76
storage queue,
compared to port files 8-1
sending and receiving data 5-76
see also STOQUE type statements 5-76
STORE statement,
definition of 5-79
syntax of 5-79
subport file attributes,
list of 8-16
subport files,
closing 8-2
opening 8-5
subports,
definition of 8-1, 8-14
requesting messages from 8-8
sending messages through 8-13
subroutine call statement,
definition of 5-80
syntax of 5-80
SUBROUTINE declaration statement,
description of 4-37
syntax of 4-37
subroutines,
calling 5-80
compared to forward procedures 4-37
subscripting 4-10
and arrays 4-10
definition of 2-7
rules for 2-8
subscripts,
and arrays 4-10
definition of 2-7
example of 2-8
function of 2-7
limits on use of 2-7
use of 2-8
suspending
a program with DATACOMM WAIT 7-16
execution of a program 5-75
program execution 5-90
suspension of program execution 8-11
syntax errors, list of messages C-1
syntax,
arithmetic operators 2-2
arrays 2-7
assignment symbol 2-2
at sign 2-10

basic symbols 2-2
braces 2-1
brackets 2-2
conditional relations 2-3
consecutive periods 2-2
controller fields 2-11
duplicate identifiers 2-6
format of programs 2-13
identifiers 2-5
key words 2-1
language statements 2-5
literals 2-10
examples of 2-10
logical operators 2-2
lower case words 2-1
non-numeric literals 2-10
notations 2-1
numeric literals 2-10
optional words 2-1
period 2-2, 11
punctuation 2-4
quotation marks 2-10
reserved words 2-4
scope of identifiers 2-6
separators 3-1
special identifiers 2-6
subscripts 2-7
underscore character 2-2
undigit numeric literals 2-10

T

table elements,
conducting tests on 5-65
searching for 5-63, 5-65
tables,
creation of B-7
definition of B-7
example of B-7
searching for an element in 5-63
tape files,
and the OPEN statement 5-55
and the READ statement 5-59
and the WRITE statement 5-94
automatic purging 4-25
specifying life of 4-25
specifying variable-length records 4-25
telephone line, disconnecting 7-6
TOPLOOP statement,
and DO statement, example of 5-37
definition of 5-81
syntax of 5-81
see also DO statement
TRACE statement,
definition of 5-82
syntax of 5-82
tracing program execution,
statement for 5-82
TRANSLATE statement,
algorithm for 5-83
definition of 5-83
example of 5-87
example process of 5-84
syntax of 5-83
translate tables,
data communications 7-15
translation
algorithm for 5-83
of strings 5-83
process, example of 5-84
table address 5-86
tables,
example of 5-87
format in memory 5-84
used with data translation 5-83
values and functions for 4-27
TRANSTBL data communications statement,
definition of 7-15
syntax of 7-15
TRUE,
and its value in a data area 4-12
truncation of data,
and data moves 3-4
type attribute,
and alignment control 3-4
types,
alphabetic 4-9
alphabetic, in an arithmetic operation 4-9
bit 4-9
double 4-9
fixed double 4-9
fixed integer 4-10
fixed real 4-10
indirect 4-10
integer 4-10
numeric 4-10
real 4-10
signed integer 4-10

U

underscore character, use of 2-2
undigit numeric literals,
definition of 2-10
function of 2-10
UNLOCK statement,

definition of 5-89
syntax of 5-89
see also LOCK
UNSEGMENTED declaration statement,
description of 4-38
syntax of 4-38
unsigned numeric literal,
controller field override 2-11
UNTIL statements,
and DO 3-2, 5-34
example of 3-2
USASCII character set, table of F-1
use routines,
allowable types 4-28
and the reader/sorter 9-1, 10-1
definition of 4-28
differences from procedures 4-28
example of 4-30
exiting 5-47
functions of 4-28
user routines,
when they are entered 4-29

V

value parameters,
definition of 4-34
value statement,
meaning of bits in 11-5
variable-length records, specifying 4-25
variables,
making them segment relative 4-9
redefining with ADDRESS 4-2

W

WAIT data communications statement,
definition of 7-16
syntax of 7-16
WAIT statement,
definition of 5-90
examples of 5-91
for port files 8-11
examples of 8-12
syntax of 8-11
syntax of 5-90
warning messages,
how to read C-1
list of C-1
WHILE statement,
and DO 3-3
definition of 3-3, 5-92
example of 3-3
see also DO statement
WHILE, and DO 5-34
work area,
specifying name for 4-24
work files,
creating 4-24
naming conventions 4-24
WORKAREA option,
specifying ON or OFF 4-24
WRITE data communications statement,
definition of 7-17
syntax of 7-17
WRITE statement,
and automatic unlock 5-94
definition of 5-93
for port files 8-13
and OPEN 8-13
examples of 8-13
syntax of 8-13
syntax of 5-93
use of LOCK with 5-93
use of OPEN with 5-93
write to control operation,
requesting 7-17
WRITEREAD data communications statement,
definition of 7-18
syntax of 7-18
WRITEREADTRANS data communications statement,
definition of 7-19
syntax of 7-19
WRITETRANSREAD data communications statement,
definition of 7-20
syntax of 7-20

Z

ZIP statement,
and significance of the period 5-95
definition of 5-95
syntax of 5-95